INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Integrated Modular Avionics for Space Applications:

Input/Output Module

Cláudio David Cardoso da Silva

Dissertação para a obtenção de Grau de Mestre em

Engenharia Aeroespacial

Júri

Presidente:	Prof. João Manuel Lage de Miranda Lemos
Orientador:	Prof. Agostinho Rui Alves da Fonseca
Vogais:	Prof. José Raúl Carreira Azinheira

Outubro 2012

Ao meu Pai...

Acknowledgements

Em primeiro lugar gostaria de agradecer aos membros do centro de excelência IMA da GMV, sem os quais esta tese não teria sido possível. Gostaria de deixar um especial agradecimento ao Eng. Tobias Schoofs, não só pela oportunidade que me deu em desenvolver esta tese em colaboração com a GMV, mas também pelas relevantes discussões técnicas.

Agradeço também ao meu orientador, Prof. Agostinho Fonseca, pelo seu suporte e ajuda na revisão desta dissertação.

Da mesma forma, gostaria de deixar o meu obrigado a todos os amigos e colegas que, de uma forma ou de outra, partilharam comigo esta etapa.

Devo um especial agradecimento ao meu colega e amigo Duarte Donas-Boto, por ter tornado o caminho até aqui consideravelmente mais simples.

Em último lugar, mas mais importante, deixo aqui um enorme agradecimento ao meu pai, a minha mãe e ao meu irmão; o seu suporte incondicional e permanente tornou isto possível.

A todos muito obrigado,

Cláudio Silva

Abstract

The European Space Agency's (ESA) IMA for Space project (IMA-SP) defines a reference architecture for space on-board software based on the concept of Integrated Modular Avionics (IMA). This novel software architecture leverages upon the success of its aeronautical IMA counterpart, which changed radically the software and hardware role in avionics development. Part of the IMA-SP platform is a generic I/O module that provides input/output services based on network interfaces and other hardware devices. In compliance with the overall objectives of the platform, the I/O component should not only integrate device control into a partitioned environment, but shall also relieve applications from the burden of addressing I/O directly.

This dissertation proposes, discusses and compares several possible architectures for the I/O module. The state of the art aeronautical avionic architecture, as well as the current and new space on-board reference platforms are described in detail to potentiate a representative comparison of the possible I/O architectures. It was concluded that the most fit solution foresees the encapsulation of all I/O related tasks in one dedicated system partition.This solution, named I/O Partition, is able to cope with the I/O handling requirements of the IMA-SP platform. However, it is subject to time partitioning and, therefore, introduces a considerable amount of latency in the I/O communication; latency that must be taken into account during application development. In the course of the implementation and testing activities, it was found that the I/O module configuration and analysis is a complex task that will require specific tool and modeling support. On the long term, it is possible to improve the performance of the developed solution through the use of multi-core computer architectures or through the development of partitioning aware I/O devices.

Keywords: IMA; Time and Space Partitioning; ARINC 653; Operating Systems; Real Time systems; Spacewire; MIL-STD-1553B; LEON3;

Resumo

O projecto *IMA for Space* (IMA-SP) da Agencia Espacial Europeia (ESA) define uma arquitectura de referência, baseada no conceito de Aviónica Modular Integrada (IMA), para o *software* embarcado no domínio espacial. Esta nova arquitectura de *software* é motivada pelo sucesso da implementação de IMA no sector aeronáutico. Neste sector, IMA revolucionou a relação entre *software* e *hardware* no desenvolvimento de sistemas aviónicos. Faz parte da plataforma IMA-SP um módulo genérico de entrada/saída (E/S) que fornece serviços de E/S baseados em interfaces de rede e em outros dispositivos de *hardware*. Em cumprimento dos objectivos gerais da plataforma, o componente de E/S não deve integrar unicamente o controlo de dispositivos num ambiente particionado mas também remover completamente a necessidade das aplicações executarem tarefas de entrada/saída.

Esta dissertação propõe, discute e compara diversas arquitecturas possíveis para o módulo de entrada/saída. O estado da arte das arquitecturas aviónicas aeronáuticas, tal como as plataformas de referência do domínio espacial são descritas em detalhe para potenciar uma comparação compreensiva das possíveis arquitecturas de E/S. Conclui-se que a solução mais adequada prevê o encapsulamento de todas as tarefas relacionadas com E/S numa partição de sistema dedicada. Esta solução, chamada de Partição de E/S, é capaz de cumprir com os requisitos de entrada/saída da plataforma IMA-SP. Contudo ao ser sujeita a particionamento temporal, especifico do executivo, introduz latência considerável nas comunicações. Esta latência deve ser tida em conta durante o desenvolvimento das aplicações. No decurso das actividades de implementação e teste da solução, foi concluído que a configuração e análise do módulo de E/S são tarefas complexas que irão necessitar de suporte de ferramentas e possivelmente de modelagem. A médio prazo é possível melhorar o desempenho da solução implementada através do uso de arquitecturas *multi-core* ou de dispositivos de entrada/saída que consigam interagir directamente com o particionamento espacial.

Palavras-chave: Aviónica Modular Integrada; Particionamento no tempo e no espaço; ARINC 653; Sistemas Operativos; Sistemas em tempo real; Spacewire; MIL-STD-1553B; LEON3

Contents

List of Tables xiii
List of Figures xvi
Acronyms xix

1 Introduction **1**
1.1 Institutional Context and Objective 2
1.2 Outline 2

2 Aeronautical Avionic Systems **3**
2.1 Historical perspective 3
2.2 Integrated Modular Avionics 4
2.2.1 Time and Space Partitioning 5
2.2.2 ARINC 653 6
2.3 Data Buses 8
2.3.1 ARINC 429 8
2.3.2 ARINC 664 - AFDX 8
2.3.3 Complementary data buses 10
2.3.4 Optical Fiber 10
2.4 I/O Handling in IMA 10
2.5 Benefits of implementing IMA 11
2.6 Section Summary 11

3 Avionic Systems in Space **13**
3.1 Space Constraints 13
3.2 Space Hardware 13
3.2.1 Data Buses 15
3.2.1.1 RS232 15
3.2.1.2 Controller Area Network 16
3.2.1.3 MIL-STD-1553B 16
3.2.1.4 SpaceWire 17
3.3 Space Software 17
3.4 Limitations of the Space Platform 18

3.5 Section Summary . . . 19

4 Integrated Modular Avionics for Space **20**

4.1 IMA-SP Platform Software . . . 20
4.2 IMA-SP Platform Hardware . . . 23
4.3 IMA-SP Platform I/O Interfaces . . . 23
4.3.1 AMBA . . . 24
4.3.2 RS232 - APBUART . . . 25
4.3.3 Spacewire - GRSPW2 . . . 26
4.3.4 MIL-STD-1553B: B1553BRM and GR1553B . . . 27
4.4 Section Summary . . . 29

5 Input/Output solutions for the IMA-SP architecture **30**

5.1 I/O segregation in partitioned systems . . . 30
5.2 Integration of I/O into the IMA-SP platform . . . 31
5.2.1 Software based solutions . . . 31
5.2.1.1 Integrated in the operating system . . . 31
5.2.1.2 Integrated in a dedicated partition . . . 33
5.2.2 Hardware based solutions . . . 35
5.2.2.1 MIL-STD-1553B Subaddressing . . . 35
5.2.2.2 AFDX in space . . . 36
5.2.2.3 Parallel I/O Component . . . 37
5.2.2.4 Enhancing I/O with RMAP . . . 38
5.2.2.5 GRSPW2 . . . 39
5.2.3 Comparison of solutions . . . 39
5.3 Section summary . . . 41

6 Proposed Solution: I/O Partition Design and Implementation **42**

6.1 I/O Component Overview . . . 42
6.2 I/O Partition Design . . . 44
6.3 I/O Partition architecture . . . 46
6.3.1 Interfaces . . . 46
6.3.2 Internal Design . . . 47
6.4 Configuration . . . 51
6.5 The MIL-STD-1553B challenge . . . 52
6.5.1 Proposed solution . . . 53

7 I/O Partition Testing **56**

7.1 Functional Testing . . . 56
7.2 Preliminary Benchmarks . . . 58

8 Conclusion **60**
8.1 Overview . . . 60
8.2 Dissemination . . . 61
8.3 Current State . . . 61
8.4 Future Work . . . 62

Bibliography **63**

A - Appendix - I/O Partition User Manual **67**
A.1 I/O Partition Concepts . . . 67
A.2 I/O Partition Configuration . . . 68
A.2.1 Generic Configuration . . . 69
A.2.2 Routing Configuration . . . 69
A.2.3 Routing Configuration Example . . . 71
A.2.4 Device Specific Configuration . . . 73
A.2.4.1 MIL-STD-1553B BC Specific Configuration . . . 73
A.2.4.2 SpaceWire Specific Configuration . . . 77
A.2.5 I/O Partition Interfaces . . . 78
A.2.5.1 Middleware Interface . . . 78
A.2.5.2 MW library configuration interface . . . 78
A.2.5.3 I/O partition MW configuration . . . 79
A.2.5.4 MW library user interface . . . 79
A.2.5.5 MW library user interface example . . . 80
A.2.5.6 Remote Ports Interface . . . 81
A.3 I/O Partition Build and Integration . . . 82

List of Tables

5.1 Comparison of proposed solutions for I/O in the IMA-SP architecture 40

7.1 Communication channels used in the Ethernet demonstrator. 57

A.1 *logical_routing_t* structure members used to define a route 70
A.2 *physical_routing_t* structure members used to define a route. 70
A.3 Command Control Word definition 74
A.4 AS1553B configuration options 75
A.5 GR1553BRM configuration options 76
A.6 GRSPW configuration options 77
A.7 Middleware configuration options 78
A.8 MW port configuration 79
A.9 *io_init* detailed interface 79
A.10 *send_read_request* detailed interface 79
A.11 *get_reply* detailed interface 80
A.12 *send_data* detailed interface 80
A.13 Remote Port Configuration 82

List of Figures

2.1 Line Repleceable Unit - LRU 4
2.2 Comparison Between the functionality of different aircrafts and the number of onboard electronic equipment 4
2.3 A simple example confronting IMA and federated architectures 5
2.4 Architectural overview of an ARINC 653 system composed by four partitions each one with its own processes 7
2.5 Partition Schedule example: Partition 1 has a period of 40ms and duration of 20ms; the remaining partitions have a period of 120ms and duration of 20ms 7
2.6 An AFDX frame. The frame is identical to a generic UDP/IP/Ethernet frame with the exception of the Sequence Number which is specific of AFDX 9

3.1 Example hardware architecture depicting a Leon 3 based onboard computer connected to several components through MIL-STD-1553B and SpaceWire data buses 14
3.2 RTEMS architecture 18

4.1 The IMA-SP executive composed by a partitioning kernel (hypervisor) and several partitions, each one running its paravirtualized version of RTEMS 21
4.2 Comparison between aeronautical and space time and space partitioned platforms 22
4.3 Exterior appearance of a RASTA system with connectors for different buses. 23
4.4 Plug and Play configuration layout for an AHB unit 24
4.5 APB slave Plug & Play Table 25
4.6 UART registers 25
4.7 Bus Controller Command Block 28

5.1 Example of an I/O module integrated in the Hypervisor 32
5.2 Diagram representing the I/O module running in a dedicated partition 33
5.3 Delay in communication originated from the I/O partition being subject to the partition schedule 34
5.4 Architectural representation of a system with the I/O module detached in a second CPU core 37

6.1 I/O component internal architecture 43

6.2 I/O component architecture when dedicated to a single partition 44
6.3 I/O partition internal task schedule 48
6.4 IOP internal I/O Cycle 50
6.5 I/O Configuration toolchain 51
6.6 Example MIL-STD-1553B and application schedule synchronization 53
6.7 A MIL-STD-1553B command list 54
6.8 Concept of Elastic Partitions 55

7.1 MIL-STD-1553B benchmark 58

A.1 Application partition communicating with I/O partition. 67
A.2 Interface with I/O partition through remote ports. 68

Acronyms

AFDX Avionics Full-Duplex Switched Ethernet

AHB Advanced High-speed Bus

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

APEX Application Executive

API Application Programming Interface

ARINC Aeronautical Radio, Incorporated

BC Bus Controller

BC Bus Controller

BM Bus Monitor

CAN Controller Area Network

COTS Commercial Off-The Shelf

CPIOM Core Processing Input/Output Module

CPM Core Processing Module

CPU Central Processing Unit

DAL Design Assurance Level

DMA Direct Memory Access

ECSS European Cooperation on Space Standardization

ESA European Space Agency

FDIR Fault Detection Isolation and Recovery

FIFOs First In First Out

FPGA Field Portable Gate Array

FSM Finite State Machine

HMI Human Machine Interface

I/O Input/Ouput

IMA Integrated Modular Avionics

IMA-SP Integrated Modular Avionics for Space

IOM Input/Output Modules

IOP I/O Partition

IP Intellectual Property

IP Internet Protocol

LRU Line Repaceable Unit

MAC Media Access Control

MIW Message Information Word

MMU Memory Management Unit

MTF Major Time Frame

MW middleware

OS Operating System

POS Partition Operaing System

RASTA Reference Avionics System Testbench Activity

RDC Remote Data Concentrator

RMAP Remote Memory Access Protocol

RT Remote Terminal

RT Remote Terminal

RTEMS Real Time Executive for Multiprocessor Systems

RTEMS-impr RTEMS Improvement

SoC System-on-Chip

SpW SpaceWire

TSP Time and Space Partitioning

UART Universal Asynchronous Receiver Transmitter

UDP User Datagram Protocol

VHDL VHSIC hardware description language

VL Virtual Link

Chapter 1

Introduction

During the last two decades the architecture behind avionics development has changed considerably. The federated architecture that was used up to the end of the century is being replaced by the Integrated Modular Avionics (IMA) concept.

In a federated architecture, a complex system is composed by smaller black boxes that provide functions. Each black box – often called Line Replaceable Unit (LRU) - contains the hardware and software required for it to provide its function. With the federated concept each new function added to an avionic system will require the addition of one of these LRUs. This fact brings an overhead in terms of volume, mass, cost and power consumption that started constraining the development of modern avionic systems. The aerospace industry understood that the classical concept of "one function = one computer" could no longer be maintained.

To solve this problem the IMA concept emerged from the Aviation Industry. IMA allowed retaining some advantages of the federated architecture, like fault containment, while decreasing the overhead of separating each function physically from others. The main architectural principle behind IMA is the introduction of a shared computing resource that contains functions from several LRUs. The problems created by sharing resources among applications were dealt by a standardization effort that led to the ARINC 653 specification.

ARINC 653 - Avionics Application Standard Software Interface – defines a standard layer between a safety critical application and the underlying Operating System (OS). The architectural backplane of this standard is the time and space partitioning (TSP) of the user's applications that prevents unwanted interference between applications.

The IMA concept matured enough and is currently being used in new aircrafts like the Airbus A380 and the Boeing 787. This adoption confirmed the projected advantages brought by the IMA concept. The IMA concept proved to meet the performance, reliability and flexibility requirements for highly integrated avionic systems as desired [1].

The success of the IMA implementation on the aeronautical domain caught the attention of the Space Industry, mainly of the European Space Agency. Almost all constraints that were positively affected by the IMA concept, like the reduction in mass and power consumption, are of acute importance to any

space platform. Furthermore, IMA facilitated the process of integrating software and hardware from different sources into one common background platform. In space, uncorrelated software from different sources shares the same hardware making the process of software integration difficult, hazardous and costly.

Given the space domain requirements and the advantages added by the IMA concept it seems natural to spin off IMA from the aeronautical domain into space. Following this reasoning, the European Space Agency has been funding a few projects which study this conceptual transposition. These projects delimitate the architectural differences between the two domains and attempt to find a tailored version of the IMA concept which is applicable to space [2].

1.1 Institutional Context and Objective

This thesis is derived from my collaboration with GMV in the scope of the ESA's Integrated Modular Avionics for Space (IMA-SP) project.

The IMA-SP project intends to adapt the IMA concept to a space platform and produce a working prototype of a typical satellite configuration but following the derived IMA concept [3, 2]. GMV's main role on the IMA-SP project is to investigate and propose input/output handling strategies for an IMA-SP system [2]. I/O handling is one of the main problems that needs to be addressed before the introduction of this new architecture in real space avionic systems. The main objective of this thesis will be the study, design, implementation and evaluation of an I/O solution for the new IMA-SP platform architecture.

1.2 Outline

In Chapter 2 I will start by introducing the Integrated Modular Avionics concept, its supporting standards and the state of the art concerning aeronautical avionic systems. I will continue, in chapter 3, by presenting the typical space platform in terms of software and underlying hardware. Chapter 4 will introduce the proposed IMA-SP platform in detail, in terms of software and hardware, with special focus on the characteristics affecting I/O handling. In chapter 5 several possible solutions for I/O module will be presented, discussed and compared. The architecture behind the implementation of the proposed I/O solution will be presented in Chapter 6. Chapter 7 will describe the testing of the implemented I/O solution. Finally, Chapter 8 will conclude and close this thesis.

Chapter 2

Aeronautical Avionic Systems

In this chapter the current state of the art avionic system will be discussed. I will describe this system from a historical perspective explaining briefly the evolution that avionic systems took to their current state. Several underlying concepts to the avionics platform will be presented.

2.1 Historical perspective

Avionics is a term used to describe electronic systems used in aviation. The word itself is result of a blend between the words “aviation” and “electronics”. The first use of electronic equipment in aircraft dates back to the Second World War, where simple radars and radios were incorporated in military aircraft. Since then the number and complexity of avionic systems has grown exponentially.

From its origin to the end of the twentieth century the avionics development has followed a federated architecture. In this architecture each function of the avionic system is a self contained black box that includes every resource needed for it to achieve its functionality. This black box, a Line Replaceable Unit, contains all the Hardware, Software and interfaces required for that function. The system integrator is unaware of the inside structure of the LRU, because its development is usually performed by an independent contractor [1].

The self-contained nature of the LRUs allows them to be modular and quickly replaceable at an operating location. This modularity decreases maintenance costs since a damaged LRU can be replaced quickly by a new stocked one. Furthermore a LRU ensures fault containment given that a falling unit will just stop providing its function and will not affect others. LRUs are designed following a set of standards, mainly ARINC specifications, which specify their interfaces and physical characteristics. Several manufacturers and military organization have also defined their own proprietary standards. Figure 2.1 depicts an Line Replaceable Unit.

As the number of functions required by an aircraft started growing, the shortcomings of the Federated Architecture became self-evident. Each new function will add a new LRU to the aircraft, requiring more power, mass, and volume from the aircraft. This direct correlation between number of functions and the mass/power consumption enforces aircrafts with high number of functions to be impractical [1]. The

Figure 2.1: Line Repleceable Unit - LRU[4]

aviation industry understood that it had to dismiss this black box architecture and embrace the sharing of physical resources among different functions. The first step towards the sharing of resources was given during the project of the Boeing 777 with the Integrated Modular Avionics concept.

Figure 2.2 depicts the growth in functionality and number of electronic equipment for different historical airplanes. When comparing the Airbus A380 with is predecessor, the A340, the number of LRUs has remained constant whereas the functionality has grown by one order of magnitude. This disparity is explained because the A380 follows an IMA concept while the A340 a federated architecture.

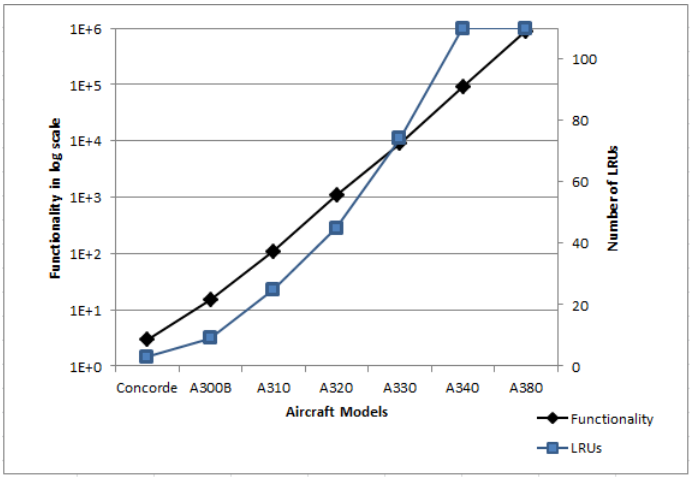

Figure 2.2: Comparison Between the functionality of different aircrafts and the number of onboard electronic equipment[4]

2.2 Integrated Modular Avionics

The Integrated Modular Avionics concept emerged as an opposing concept to the federated architecture, and offered the possibility of integrating multiple functions into the same set of physical resources. IMA identifies the set of resources that are common to almost all onboard functions. These resources were previously replicated in every LRU and are now shared among different functions. IMA architectures provide a shared resource pool composed by processing, power, communications, and I/O modules that are partitioned for use by multiple avionics functions.

In IMA, the processing capabilities are provided by Core Processing Modules (CPM). A CPM provides a common pool of processing resources that are used by different user applications that may

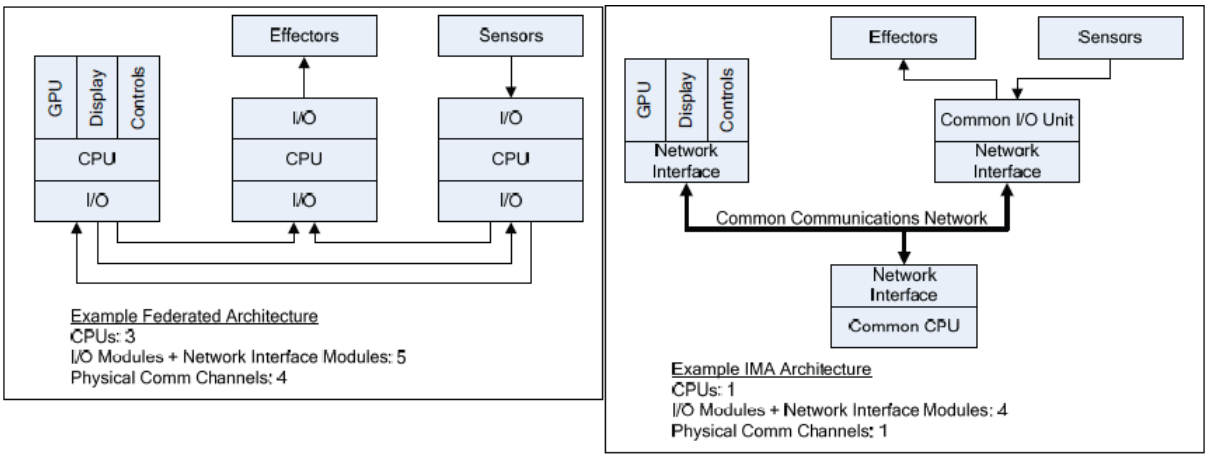

Figure 2.3: A simple example confronting IMA and federated architectures [5]

belong to different subsystems, may be supplied by different vendors and may have distinct criticality levels.

Multiple function integration on one processing module requires specific provisions in the form of requirements that are not relevant for regular LRUs. The physical separation that existed between LRUs has now to be provided virtually for applications running in the same CPM. Furthermore, the sharing of the same computing resource influences the development process, because new dependencies appear among different elements. These new challenges brought by the resource sharing concept were tackled by several standards that will be discussed in the following sections.

Input/output common resources are provided by I/O modules. These I/O Modules (IOMs) interface with sensors/actuators, acting as a bridge between them and CPM's. A Core processing module that also contains I/O capabilities is named a Core Processing Input/Output Module (CPIOM). CPIOMs are the backbone of modern avionics systems.

Reference [5] gives an example, depicted in Figure 2.3, on how an IMA architecture can benefit an avionic system. A simple graphical user interface which interacts with sensors and actuators will occupy three CPUs in the federated architecture against only one when following an IMA concept. The impact in the communication channels is also visible, reducing from five communication channels in the federated example against one in IMA.

2.2.1 Time and Space Partitioning

In the traditional federated architecture, software was an invisible part of black-boxed components which in their turn were the building blocks of avionics systems. IMA changed the software role since different applications must now share the same computing resources. The physical separation induced by the LRUs on the software must be virtually enforced in an IMA platform. The performance of each application shall be unaffected by presence of others. This separation is provided by partitioning the common resources and assigning those partitioned resources to an application [6].

The partitioning of processing power is enforced by strictly limiting the time each application is allowed to use the processor and by restraining its memory access. Memory access is controlled by

hardware thus preventing partitions from interfering with each other. Each software application is therefore partitioned in space and time [7].

Avionic software applications have different levels of criticality based on the effects that a failure on a given application would cause to the system. Those criticality levels are specified in standards like the RTCA/DO178B [8] which defines five different criticality levels. The software development efforts and costs grow exponentially with the criticality level required, since the process of testing and validation becomes more complex.

In a non-partitioned architecture all the software has to be validated under the same criticality level, which is the highest among the applications that are being integrated [8]. This is required because a less critical application (e.g. the air conditioning system) cannot influence an application with a higher criticality (e.g. flight controls). The result of such an influence could be a catastrophic failure. Robust time and space partitioning enables software with different critically levels to be integrated under the same platform, easing the integration and certification process, since it prevents all interference between applications [3].

Since each application is isolated both in time and in space from others it is guaranteed that any fault will not propagate. As a result it is possible to create an integrated system that has the same inherent fault containment as a federated system. Additionally, the strict resource segregation opens the possibility of separate verification/validation of hosted applications within the shared computing resource [3].

2.2.2 ARINC 653

Following the previous reasoning it is clear that TSP is one of the key elements in the IMA software infrastructure. The time and space partitioning architecture used in IMA was standardized in an effort shared across all major players in the aviation industry. The standard resulting from this effort was the ARINC specification 653 [9, 10].

ARINC 653 [9] defines a standard interface between the software applications and the underlying operating system. This middle layer is known as application executive (APEX) interface. The philosophy of ARINC 653 is centered upon a robust time and space partitioned operating system, which allows independent execution of different partitions. In ARINC 653, a partition is defined as portions of software specific to avionics applications that are subject to robust space and time partitioning [11, 12]. They occupy a similar role to processes in regular operating systems, having their own data, context, attributes, etc.

The underlying architecture of a partition is similar to that of a multitasking application within a general-purpose computer. Each partition consists of one or more concurrently executing processes (threads), sharing access to processor resources based upon the requirements of the application [9]. An application partition is limited to use the services provided by the APEX defined in ARINC 653 while system partitions can use interfaces that are specific to the underlying hardware or platform. A partition usually hosts the software that was, in the Federated Architecture, part of one LRU. The generic

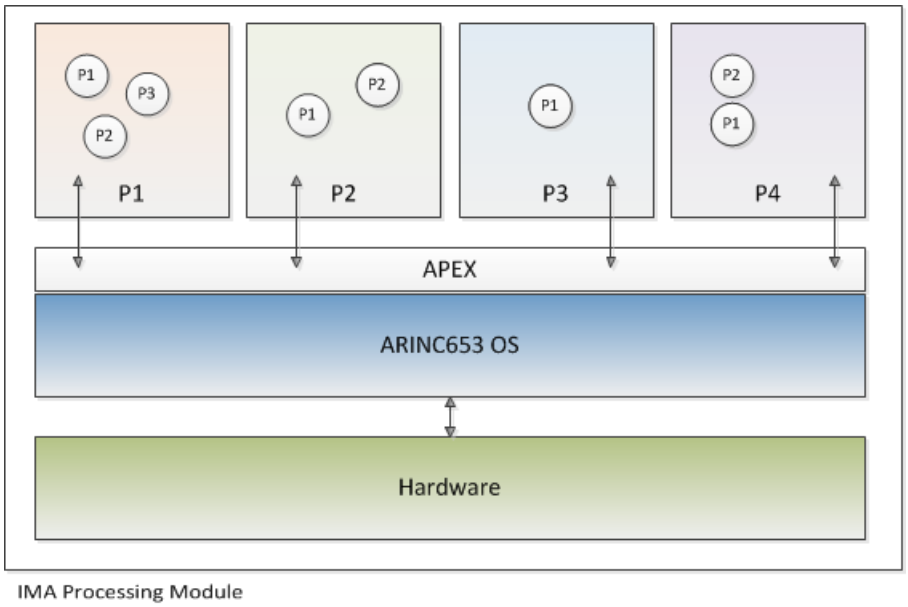

Figure 2.4: Architectural overview of an ARINC 653 system composed by four partitions each one with its own processes

architecture of an ARINC 653 operating system with four partitions is depicted in Figure 2.4.

Each partition is scheduled to the processor in a fixed, predetermined, cyclic basis, guaranteeing temporal segregation. The static schedule is defined by specifying the period and the duration of each partition's execution. The period of a partition is defined as the interval at which computing resources are assigned to it while the duration is defined as the amount of execution time required by the partition within one period. The periods and durations of all partitions are used to compose a major time frame. The major time frame is the schedule basic unit that is cyclically repeated. Figure 2.5 depicts the major frame time of a simple schedule for four partitions.

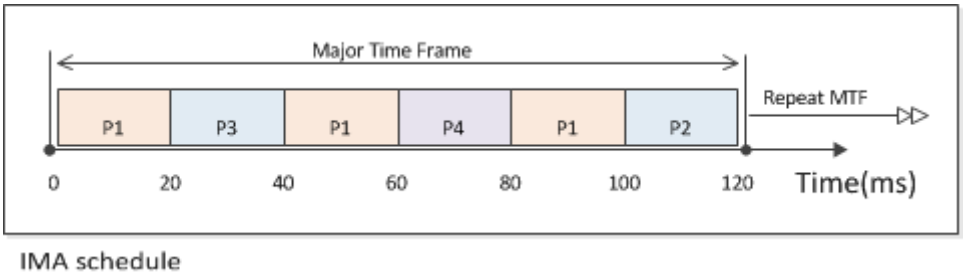

Figure 2.5: Partition Schedule example: Partition 1 has a period of 40ms and duration of 20ms; the remaining partitions have a period of 120ms and duration of 20ms

Each partition has predetermined areas of memory allocated to it. These unique memory areas are identified based upon the requirements of the individual partitions, and vary in size and access rights [9]. The standard does not specify how the memory segregation shall be accomplished, but requires it [13].

The APEX provides services that can be grouped into the following major categories [11]:

- Partition Management;

- Process Management;
- Time Management;
- Memory Management;
- Interpartition Communication;
- Intrapartition Communication;
- Health Monitor;

In[13] a detailed overview of these services is presented. It is especially noticeable that ARINC 653 does not specify services that are directly linked with I/O (like the POSIX read and write) [14]. All communication destined to entities outside the partition is performed through ports or using shared memory areas. Each port may be configured to operate in one of two possible modes. In the sampling mode, successive messages typically carry identical but updated data. Each new instance of a message overwrites the current message. In the queuing mode, each new instance of a message may carry uniquely different data and is not allowed to overwrite previous ones during the transfer [9].

2.3 Data Buses

A new communication data bus had to be developed to support the communication between the several IMA modules. The predominant data bus in the aviation industry, ARINC 429, was developed over 40 years ago and doesn't have the capabilities to support a modern IMA network.

2.3.1 ARINC 429

ARINC 429 is a self-clocking self-synchronizing data bus, with a network topology of one-to-one or one-to-many and that can achieve data rates up to 100 Kbit/s [15]. These features are inadequate in an IMA system because each new addition to the network will require adding new cabling to connect the new node with its different interfaces. Each pair of interconnected LRUs requires a cable interconnecting them, which impacts the total weight and volume. Additionally the bandwidth provided by this data bus is not sufficient to fulfill the demands of modern avionic networks.

Several new data bus standards were proposed to replace ARINC 429. The ARINC specification 664 was pushed by Airbus during the A380 program and is becoming the de-facto standard for avionic data networks.

2.3.2 ARINC 664 - AFDX

During the A380 program, Airbus developed a new data communication network named AFDX – Avionics Full-Duplex Switched Ethernet. AFDX was developed using commercial off-the shelf (COTS) components and standards, but aiming to provide the same levels of reliability and robustness found in earlier data buses. ARINC later standardized AFDX in the ARINC 664 specification.

7 bytes	1 byte	6 bytes	6 bytes	2 bytes	20 bytes	8 bytes	17 to 1471 bytes	1 byte	4 bytes	12 bytes
Preamble	Start Frame Delimiter	Destination Address	Source Address	Type IPv4	IP Structure	UDP Structure	AFDX Payload	Seq Number	Frame Check Seq	Interframe Gap

Figure 2.6: An AFDX frame. The frame is identical to a generic UDP/IP/Ethernet frame with the exception of the Sequence Number which is specific of AFDX [14]

AFDX is a serial data transfer method based on conventional Ethernet which is defined in the standard IEEE802.3. It can achieve speed up to 100 Mbits/s using an optic fiber or copper wire backbone [16]. The use of Ethernet implied a low cost and high speed network that could be built using already widely available technologies and standards. However, regular Ethernet had to be extended to ensure the deterministic behavior and high reliability required for an aircraft data network.

Since it is full duplex, AFDX allows bidirectional high speed communication without any possibility of collision between data packets. Also as a means to ensure the characteristic determinism of point-to-point networks, AFDX provides quality of service and bandwidth reservation services which guarantee jitter and maximum end-to-end latency in the network. Bandwidth reservation is a key feature since it enables different applications to share the network without possibility of interference. To increase the robustness of the system a redundant network is used and data is sent simultaneously to both networks.

The core aspect of AFDX is the concept of Virtual Link (VL). A VL defines a virtual unidirectional point-to-point connection in the network. It provides a logical path between one source node and several destination nodes. Bandwidth reservation is accomplished by assigning each VL with a given bandwidth. VLs are also used for routing, occupying the role of the MAC address in regular Ethernet networks [16].

Switches are responsible for ensuring that frames are not lost and that each VL stays within its allocated bandwidth. The switch must reject any erroneous data, maintaining the determinism of the network. Each switch is loaded with a configuration table that links the allocated VLs in the network with one of the switch's output ports.

VLs are assigned to mailboxes on the sending and receiving end systems, and mailboxes are assigned to partitions. This way, applications are able to access the AFDX memory directly without causing any harm to the AFDX device itself or to other partitions. AFDX is, hence, tightly coupled with an ARINC 653 OS.

AFDX also uses a User Datagram Protocol (UDP) and Internet Protocol (IP) stack for sub-routing and data validation. IP is used to fragment data packets which are bigger than the maximum transmission unit of the network (1500 bytes). Fragmented packets are then reassembled in the destination, using the IP protocol fragmentation features. A Sequence Number (SN) byte is added to the IP protocol to guarantee that no packets are lost or duplicated in the network. UDP is used to select the destination port in the target application and to verify the integrity of the user's data. Figure 2.6 depicts an AFDX frame as it is transmitted through the network.

2.3.3 Complementary data buses

The main data bus is generally complemented by other buses in some subsystems. These other buses are used to interface sensors/actuators or to implement subsystems where the characteristics of the main bus are not appropriate for use.

Some of the most used complementary data buses are the MIL-STD-1553B, Time Triggered Protocol and the Controller Area Network.

2.3.4 Optical Fiber

Optic fiber is expected to replace copper wiring in avionic data buses somewhere in the next decades. Optical fiber weights only a fraction of the copper wire weight per meter and has increased safety since it is not affected by electromagnetic interference. Complex modulation techniques can be applied to optical fiber networks increasing the data rate and ensuring bandwidth allocation through the use of multiple channels in one fiber.

As of today one of the greatest obstacles to the adoption of optic fiber is that there are no standardization and certification means that are comparable to the strict requirements for on-board systems and software. Also, there is little experience with optical fiber in relation to abrupt temperature or pressure changes. Additionally the current implementation of fiber networks in ground systems differs considerably from the one expected on an aircraft.

2.4 I/O Handling in IMA

The sharing of I/O devices needs special attention since it is very dependent on the aircraft configuration and on the device itself. There are devices and buses which are specially developed to be included in an IMA system. Examples of such are AFDX devices that make use of mailboxes and VLs to interact directly and autonomously with each partition requiring I/O [17].

For devices which are not IMA ready, I/O is expensive, complex, hardware-dependent and error-prone. I/O is expensive not only due to the time it requires to perform its tasks, but also in the sense that it is difficult to control and to predict.

Regular operating systems hide the custom code which controls devices into their kernels and provide higher level interfaces for the applications to interact with those devices. The same architecture cannot be applied to a robust partitioned OS since it may violate spatial partitioning. If devices and other resources are directly shared across partitions then there is a risk of the devices being left in an unknown state following a partition error [17].

The most common solution for this I/O problem is to place the driver software in a dedicated partition which manages the device and operates it [18]. If more than one partition needs the device, the management partition acts as a router which distributes the device's data among the requesting partitions.

Robust partitioning demands that applications use only those time resources that have been reserved for them during the system design phase. Additionally on some circumstances the OS is strictly

forbidden to interrupt a partition until its duration has ended. In consequence, it must be ensured that I/O devices have enough buffering capabilities at their disposal to store data during the time that their management partition is not running.

The solution to manage a device using a partition is not efficient since it forces I/O to be subject to the partition schedule [14]. A new generation of IMA systems uses a specific device named a Remote Data Concentrator which gathers data from different sensors and sends it to the main IMA module through AFDX or other IMA network [4]. Thus RDCs become responsible for the I/O from these devices releasing the main IMA modules for functional tasks.

2.5 Benefits of implementing IMA

Generically, the expected benefits from implementing IMA solutions are [5, 4, 3]:

- Optimizations of physical equipment Weight, Volume and Power Consumption;
- Focused development efforts: the developer can focus wholly on their software, instead of focusing on the complete development of a LRU;
- Increased market competition: small companies can specialize in providing parts of the integrated architecture. (e.g CPM, ARINC 653 OS);
- Retaining federated system properties like fault containment;
- Incremental validation and certification of applications;
- Easier integration process;

IMA has already been applied in the development of several aircrafts, most noticeably on the development of the A380 and Boeing 787. The Airbus and Boeing programs reported savings in terms of mass, power and volume of 25%, 50% and 60% respectively [4]. IMA has eliminated 100 LRUs from the Boeing 787.

2.6 Section Summary

This chapter introduced the technologies that are behind modern avionic systems, including the Integrated Modular Avionics (IMA) concept and the AFDX network.

IMA enables totally unrelated applications to share the same physical resources without impacting the safety and security of the system. To achieve this high level of integration between applications IMA foresees the use of a time and space partitioned (TSP) platform that is specified in the ARINC 653 standard. Each application instance of the TSP platform is segregated from others, occupying different memory areas and using the processor during strictly allocated timeframes. ARINC 653 defines an application programing interface that abstracts the underlying hardware and TSP platform from the applications.

A new network, AFDX, was developed to support the IMA concept of resource sharing. This high speed network is built on commercial standards and supports features needed to enable network sharing by different uncorrelated applications.

Modern IMA implementations use core processing modules (CPM) to host applications in an ARINC 653 platform. I/O functionality is kept apart in remote data concentrators (RDC) that aggregate data from transducers. CPMs, RDCs and other onboard systems are then connected through AFDX or other IMA compatible network.

The benefits of using IMA to develop avionic systems were visible during the programs of the Boeing 747 and Airbus A380. It is expected that this set of technologies will become the de facto standard for avionic system in commercial aircrafts.

Chapter 3

Avionic Systems in Space

In this chapter, the technologies that are relevant for a typical space avionic system will be introduced.

3.1 Space Constraints

There are some major differences between the space and aeronautical domains which constrain the development of space avionic systems. While most aeronautical systems operate with human intervention most space systems are unmanned. This difference is crucial since a space system cannot undertake physical maintenance operations nor be upgraded. Orbital presence is as well very demanding due to drastic temperature variations and exposition to radiation. Radiation is major threat to onboard electronic systems and software since it can cause electrical fluctuations and software errors.

Additionally space systems are very constrained in terms of available power, mass and volume. It is expensive and impractical to develop and deploy complex platforms therefore most systems have very limited hardware. Satellites are usually limited to one or two main computers connected with the required transducers and payload equipment through robust data buses.

When compared with commercial aviation the space market lacks standardization. Each major player in the industry designs and operates its systems using their own internal principles. The European Space Agency has expended a great amount of effort in the last decades to standardize European space platforms. These efforts lead to the ECSS standards and to the homogenization of the hardware and software platforms.

3.2 Space Hardware

Typical spacecraft avionics architecture is composed by two different functional systems, one is responsible for data management operations and the other for Attitude & Orbital Control [17, 3]. Some implementations separate these systems in different hardware platforms, which are generically composed by a processing unit, several auxiliary modules, specific data buses and peripherals.

The onboard computer is typically connected to sensors, actuators and other critical systems by

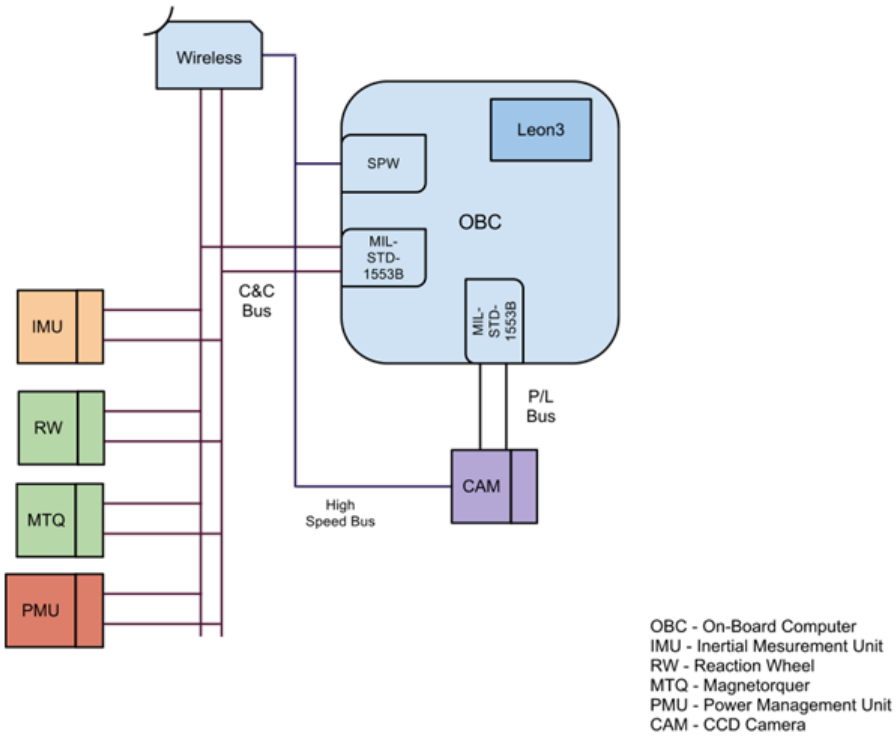

Figure 3.1: Example hardware architecture depicting a Leon 3 based onboard computer connected to several components through MIL-STD-1553B and SpaceWire data buses

means of a time predictable data bus. A second, more flexible, data bus is used to connect devices requiring high data throughput. High data throughput devices encompass antennas, mass memory units, cameras, etc. Figure 3.1 represents diagrammatically an example on-board architecture.

All critical hardware used in space is radiation hardened and ruggedized. Radiation effects (total dose, latchup, single event upsets) are one of the main concerns for space microelectronics [19]. Radiation tolerance can often be achieved purely by carefully selecting the schematics and layout of the chip. Therefore most of the higher level hardware destined to space is available in the form of intellectual property VHDL cores that can be implemented in a FPGA or other reconfigurable hardware. These IP cores are developed by ESA and several private companies and are usually available with an open source license. Between the hardware available as IP cores there are radiation hardened processors, fault tolerant memory controllers, several peripherals and data bus interfaces.

Several distinct IP cores are bundled together to create a system-on-chip (SoC). These SoC integrate all the functionality needed for a fully operational system in one FPGA, while reducing mass and power consumption. The European Space Agency promotes the use of SoCs in space, and has developed a few for use in its own missions or to accomplish specific tasks (e.g. data processing). These systems include standard processor devices, dedicated processing blocks, interfaces to various peripherals and on-chip bus structures.

One of the SoCS used and developed by ESA is the Spacecraft Controller On-Chip (SCOC3) which can act as a spacecraft main computer. SCOC3 features a LEON3-FT processor with GRFPU running at 120 MHz. The I/O subsystem provides UART, Controller Area Network, SpaceWire and MIL-STD-

1553B interfaces. SCOC3 also includes a Telecommand/Telemetry unit and a CCDCS time generator. This system on chip can be considered as containing the most common architecture currently in use for European space systems [20].

The LEON family of processors was specially developed by ESA for use in the harsh space environment. Leon processors are based on the reduced instruction set architecture SPARC whose main advantage is the use of register windows [21]. The newer versions of these processors, namely the LEON3 and LEON4, are available in Fault Tolerant (FT) versions [22]. These specially designed versions are capable of correcting single event upset errors in registers, cache and memory without impacting the instruction execution timing. The LEON3 developer, Aeroflex Gaisler, has validated a LEON3-FT processor design implemented in an RTAX2000s radiation tolerant FPGA for use in critical space applications [20].

All Leon3 and Leon4 implementations are centered on the AMBA Advanced High-speed Bus (AHB), to which the processor and other high-bandwidth units are connected. Low Bandwidth units are connected to the AMBA Advanced Peripheral Bus (APB). The Advanced Microcontroller Bus Architecture (AMBA) is a bus system introduced by ARM Ltd in 1996. AMBA was developed to be specially used in embedded systems, favoring modular system design and processor independence. It provides two distinct buses: AHB which is used for high performance, high frequency system modules and APB which is optimized for minimal power consumption and reduced interface complexity.

3.2.1 Data Buses

There are several data buses used in space avionic systems. The most common are: RS232, CAN, MIL-STD-1553B and SpaceWire.

3.2.1.1 RS232

RS232 is one of the oldest serial data buses with the first revision of the standard being published in the sixties. RS232 was designed for point-to-point connections between a data terminal and a mainframe computer and is capable of full and half duplex communication between the terminals. Until the introduction of the universal serial bus (USB) RS232 was the most used bus in personal computers. Nowadays, it still features broad adoption in embedded systems mainly due to its simplicity and interoperability.

The RS232 standard defines the electrical and mechanical characteristics of the bus, but doesn't specify the character encoding and message structure. However, practical implementations typically specify a data structure with start and stop bits, plus an optional parity bit, and a data field of 7 or 8 bits. The standard also limits the transfer rate to a maximum of 20000 bits per second but most devices can operate with higher data rates [23].

The fact that RS232 uses static voltage levels of very high value to specify each logic level (0 or 1) is a major limitation, since the signals are more vulnerable to noise and interference. Still, the simplicity and easy implementation of a serial transmitter/receiver make RS232 an excellent choice for systems with limited resources.

3.2.1.2 Controller Area Network

The controller area network (CAN) bus is a multi master message broadcast system which can achieve speeds up to 1Mbit/s. CAN was developed by BOSCH to replace multi wire networks used in the automotive industry.

The CAN standard defines a bus topology where every node is connected to pair of wires, CAN High and CAN Low, which are terminated by 120 Ohm resistors. Signaling is differential augmenting the bus immunity to noise and increasing fault tolerance. The electrical current in each wire is equal but opposite in direction resulting in a noise canceling effect. Each node requires a transceiver to communicate with the bus and a CAN controller which serializes CAN messages to the transceiver [24].

Each node in a CAN network is able to send and receive messages, but not simultaneously. Each message has a given priority and can contain up to 8 eight data bytes. CAN is thus optimized to broadcast small data messages, like sensor data. When two nodes start simultaneously transmitting a message, the message with higher priority wins the bus access relegating the lower priority messages to a later time.

Only the data link and parts of the physical layer are specified by the CAN standard. The upper communication layers are left open for the network developer. Several standards were introduced to define the upper layers of the CAN stack, among them the CANOpen and ARINC 825 standards.

3.2.1.3 MIL-STD-1553B

MIL-STD-1553B is a single-master multi-slave serial data bus standard published by the United States of America Department of Defense that was originally designed for use with military avionics. This bus is widely known because it was chosen to equip the F-16 Fighting Falcon as its main avionic data bus.

This bus features a dual redundant physical layer, balanced differential signaling and half duplex communication with the speed of 1Mbit/s. Every device connected to the bus can be in one of three possible modes: Remote Terminal, Bus Controller or Bus Monitor. Normal operation only uses the primary bus with the secondary being used in case of failure. MIL-STD -1553B has proven to be a high reliable bus with less than one word fault per ten million words [25].

A Bus Controller acts as the bus master and is responsible for initiating all communications in the bus. All information flowing to the bus is communicated using a command/reply system: the BC sends a command to the remote terminal which in its turn replies with a response. Multiple BCs can exist, but only one can be active at a time. BCs usually transmit their commands according to a predetermined fixed schedule, ensuring a time deterministic bus.

The remote terminal is a device connected to the MIL-STD-1553B bus that receives and processes commands from the bus controller, verifies for communication errors and replies to those commands. Each bus can have up to 31 remote terminals each one with a unique address. A RT terminal can only send data to the bus if it was ordered by the BC.

A Bus Monitor is a passive device which is connected to a MIL-STD-1553B bus with the sole purpose of capturing all data being transferred through the bus. This data can be later obtained for data bus

testing and analysis.

3.2.1.4 SpaceWire

The main communication protocols used in the aerospace domain were up until recently, MIL-STD-1553B and CAN, which shared some common limitations:

- Bit rate limited to 1 Mbit/s;
- Half Duplex communication;
- Single data bus with no routing capabilities;

SpaceWire (SpW) was developed by a consortium of international space agencies to overcome these shortcomings. It features a serial full-duplex bidirectional data link, with theoretical speeds up to 400 Mbit/s. Real implementations are generally limited to 200Mbit/s or slower bit rates. At character level SpaceWire is compatible with the IEEE 1355-1995 standard, with additional Time-Codes to allow support for the distribution of system time. Exchange level features like handshaking protocol and error recovery are also defined by SpaceWire. Each SpW device uses an internal finite state machine with predefined states and transitions which enables flow control, detection of disconnect errors, detection of parity errors and link error recovery among other advanced features [26].

The SpaceWire standard defines a simple packet structure composed by a destination address, a cargo and an end of packet marker. This destination address can either be a logical identifier code of the destination node, or the logical path the packet takes to reach the destination node. Packet routing based only on the destination logical identifier requires each node to have a routing table. When path addressing is used there is no need for such routing table.

RMAP

Several protocols are being developed for use on top of a SpaceWire bus but most of them are still in a development/draft phase. One of these protocols is the Remote Memory Access Protocol (RMAP) [27]. RMAP is defined in the ECSS-E-ST-50-52C standard and aims to provide a standard mechanism to allow direct remote memory access to a SpaceWire node [27]. The standard defines a set of commands that allow reading from and writing to the memory of a SpaceWire node. These direct memory access capabilities are very important in the space domain since they allow remote software patching and remote device control. RMAP commands are usually handled by a special IP core that interprets the command and makes the necessary memory operation without the target device intervention.

3.3 Space Software

As part of ESA's effort to create a standard space avionic platform several projects were funded to port already existent, free and commercial, operating systems into the Leon architecture. Most operating systems used in space are real time operating systems which were specially qualified for use in Space.

They generally feature small footprint multi-threaded environments with or without memory protection mechanisms. For some specific tasks a general purpose operating system like Linux can be used, although Linux is not space certified.

Among the operating systems that support the Leon architecture there is one open source solution – RTEMS – that is recommended by ESA for space programmes [2] . The Real Time Executive for Multiprocessor Systems (RTEMS) is a free and open source real time operating system which aims to compete with commercial platforms in the development of hard real time embedded systems. RTEMS is designed to support applications with the most stringent real-time requirements while being compatible with open standards such as POSIX or iTRON [28]. RTEMS is included in some of the most publicized ESA missions like Herschel, Planck, Lisa Pathfinder among others. Figure 3.2 depicts the current RTEMS arcitecture and its userspace Application Programming Interface.

ESA funded a project to tailor and qualify RTEMS following a Galileo Software Standards SW-DAL B level. This RTEMS-improvement version only uses the subset of RTEMS features that are required for space projects. Details about RTEMS-impr architecture and features can be found in references [29, 13, 6].

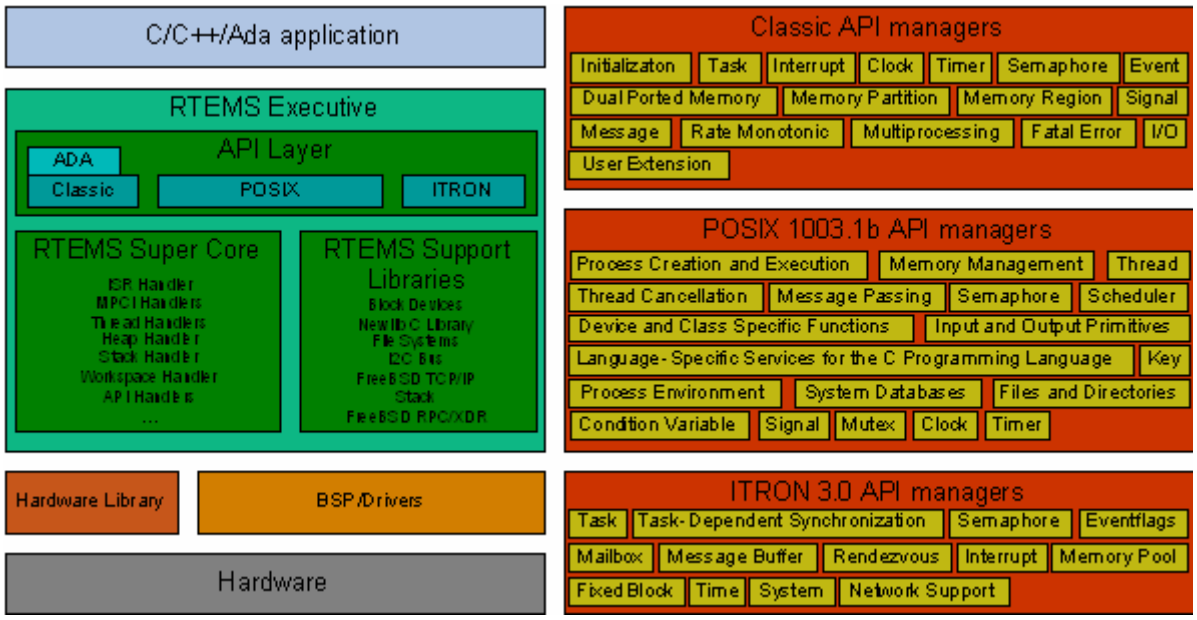

Figure 3.2: RTEMS architecture [28]

3.4 Limitations of the Space Platform

The major limitations in the current space platform are derived from the development and integration processes. Since applications from different vendors are integrated under the same operating environment they need to be certified according to the highest criticality level in the system. This process of integrating software from different sources, with different requirements and interfaces is costly and very lengthy. If the requirements and interfaces for each software module are not accurately defined in an early stage of the process it can lead to conflicts in the integration phase and to delays in the project

schedule. Additionally, in the current software platform is also possible that a small change in the software leads to extensive non-regression testing of the complete onboard software. These integration phase limitations are starting to constraint the development of complex satellites. One of the projects that has been affected by these problems is Galileo, mainly due to the high number of software modules involved.

Another disadvantage of every software module sharing the same computing resources without isolation is that security related requirements cannot be addressed. To handle these requirements additional hardware must be added to ensure complete isolation between platform software and security related functionality.

3.5 Section Summary

This section presented an overview of the elements composing the current space avionics platform. Space systems require specially developed hardware that can resist the harsh space environment and the mechanical stress generated by launch. Satellites are generally composed by two main computers: one is responsible for data handling and other for attitude and orbit control. On-board computers, transducers and any payload are connected through robust data buses. The most commonly used are MIL-STD-1553B, SpaceWire and CAN. In terms of software, space systems are based on qualified real time operating systems. One of the RTOS recommended by ESA for use in space is RTEMS. RTEMS is an open source multithreaded single process operating system which is validated and pre-qualified. The non partitioned architecture of these RTOS has some limitations since it disturbs the software integration process an increases costs.

Chapter 4

Integrated Modular Avionics for Space

This chapter will introduce the current ongoing efforts sponsored by ESA to transport the Integrated Modular Avionics concept to space systems. The goal of the current activities in this field is to demonstrate that the successful Integrated Modular Avionics (IMA) concept deployed in the civil aviation industry is also beneficial to space avionics developments [2].

As verified in previous chapters this architectural transposition is motivated essentially by the need to manage the growth in complexity of on board software. IMA's strategy of incremental software validation can have a positive impact in the effort required to validate complex systems.

In this chapter, the reference IMA-SP platform will be described in detail at software and hardware level. A special focus is given to the I/O devices since detailed knowledge of their interfaces is required to successfully discuss and implement an I/O module for the IMA-SP platform.

4.1 IMA-SP Platform Software

In order to successfully adopt the IMA for Space concept it is necessary to identify the areas where the aeronautical IMA technology is different to the space domain. These areas are identified as technology gaps which must be addressed before the IMA for Space can be demonstrated in the space systems context.

ESA has defined several ground rules which guide the IMA-SP system platform specification. They intend to adapt IMA to space without breaking with the current space avionics platform. There are several hardware and software modules already developed for the current platform that shall remain compatible with any new architecture. Failure to maintain this compatibility between modules would increase the costs of any architectural transposition. This requirement forces interfaces to remain consistent in this new IMA-SP platform. Among the interfaces that shall remain valid there is the RTEMS application programming interface upon which several onboard software depends on and the data buses which interface with available space hardware [2, 3].

Several software modules were built on top of the RTEMS OS. In order to avoid a reimplementation of these software modules it is logical to maintain RTEMS as a major component in the new IMA-SP

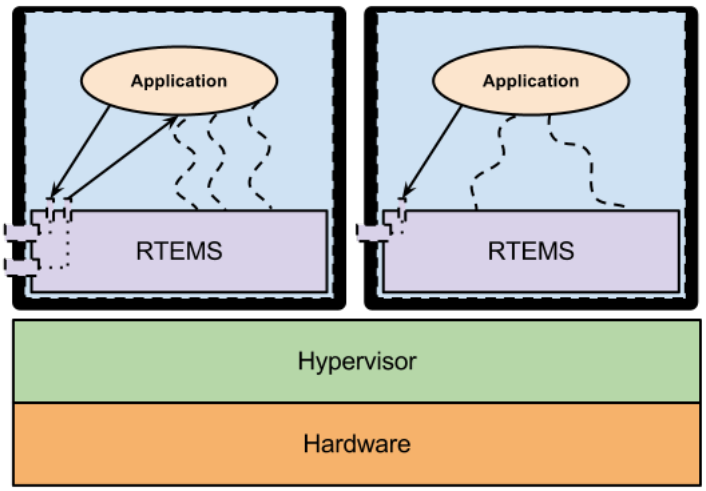

Figure 4.1: The IMA-SP executive composed by a partitioning kernel (hypervisor) and several partitions, each one running its paravirtualized version of RTEMS

platform [30]. RTEMS already provides several managers and APIs for services equivalent to the ones defined on ARINC 653, therefore the full set of an ARINC 653 service implementation, as available in aeronautical IMA, is not required.

Time and space partitioning is an essential baseline for which the IMA standards have been developed. It's crucial to maintain this feature in the IMA-SP platform. The spatial isolation of the applications can be provided by the memory management unit available in the Leon architecture. Time division multiplexing can be achieved on the underlying operating system by implementing a static partitioning schedule.

To enable the usage of RTEMS with the requirement to implement time and space partitioning, ESA defined a two level software executive. The system executive level is composed by a software hypervisor that segregates computing resources between partitions. This hypervisor is responsible for the robust isolation of the applications and for implementing the static CPU allocation schedule. A second level, the application level, is composed by the user's applications running in an isolated environment (partition). Each application can implement a system function by running several tasks/processes.

The multi-tasking environment is provided by a paravirtualized operating system which runs in each partition. These partition operating systems (POS) are modified to operate along with the underlying hypervisor. The hypervisor supplies a software interface layer to which these operating systems attach to. In the context of IMA for Space, ESA selected the use of RTEMS as the main partition operating system. A graphical representation of an IMA-SP executive is depicted in Figure 4.1.

The possible use of a virtualized operating system in each partition is a key change when compared to the ARINC 653 standard aeronautical implementation. In the aeronautical domain the TSP kernel is responsible for providing an interface directly to the user applications (APEX), besides ensuring correct isolation of resources. On the other hand the partitioning kernel in the space domain is responsible for managing partitions and for providing a virtualization interface (kernel abstraction layer) to which other

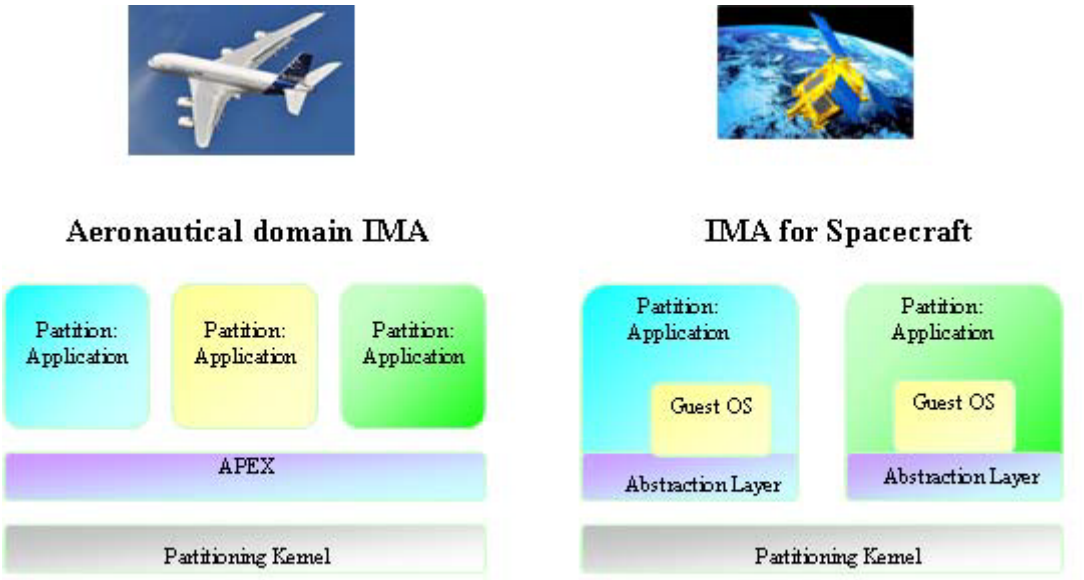

Figure 4.2: Comparison between aeronautical and space time and space partitioned platforms [31]

operating systems can attach to. In the latter case, the multitasking and intrapartition synchronization services are provided by the partition operating system. Figure 4.2 depicts the conceptual differences between the aeronautical and space TSP platforms.

Three hypervisors are currently being evaluated by ESA as part of the IMA-SP platform: XtratuM, AIR and PikeOS [32]. XtratuM is an open source hypervisor available for the x86 and Leon architectures that is developed by the Universidad Politécnica de Valencia. Despite providing similar services to the ARINC 653 standard, XtratuM does not aim at being ARINC 653 compatible [33]. PikeOS is a commercial microkernel which supports a large number of APIs and virtualized operating systems. Finally AIR is an open source operating system developed by GMV and based on RTEMS [6].

Besides maintaining the presence of RTEMS, the IMA-SP reference platform also introduces new services like cache management. A subset of ARINC 653 services is also reused. These services are mainly those which are closely related to the time and space partitioning concept and don't have a counterpart in the RTEMS service specification (e.g. Interpartition communication, partition management, health monitoring) [34]. From the developers' perspective the application programming interface available in each partition is a fusion of the RTEMS API with a subset of ARINC 653 and with some new services. The full API is available in the IMA-SP platform specification [34].

The IMA-SP specification also leaves an option to have partitions without RTEMS. These partitions can be used for very critical single threading code which doesn't requires a full featured real time operating system. The remainder of the IMA-SP API is still available to the application but with some constrains.

4.2 IMA-SP Platform Hardware

Regarding the hardware, ESA specifies the IMA-SP hardware node as being a typical LEON3 processor with memory management unit (MMU). The used input/output interfaces are MIL-STD-1553B, SpaceWire and UART. More concretely the target reference platform is a RASTA development board which is provided by Aeroflex Gaisler.

The RASTA (Reference Avionics System Testbench Activity) system was developed in order to provide a standard hardware infrastructure that allows for easier software integration. The lack of reference development systems previously to the RASTA systems prevented “many software and hardware components developed in European R&D activities to reach the required maturity level for use in space projects” [35].

The RASTA 105 board, depicted in Figure 4.3, used as reference architecture contains [36]:

- 1 LEON3 Fault Tolerant processor with MMU running at 80Mhz;
- 3 x SpaceWire interfaces – GRSPW IP core;
- MIL-STD-1553B BC/RT/BM with A/B buses – AS1553B or GR1553B IP cores;
- 10/100 Mbps Ethernet interface – GRETH IP core;
- CAN 2.0B with 2 buses – GRCAN IP core;
- 2 x 22 bit GPIO;
- 2 UART serial interfaces;
- SRAM/SDRAM/FLASH PROM memory;

Figure 4.3: Exterior appearance of a RASTA system with connectors for different buses [37]

4.3 IMA-SP Platform I/O Interfaces

The I/O interfaces in the IMA-SP reference hardware platform must be analyzed in detail to understand which of its features can be useful when implementing an I/O module or which features may compromise

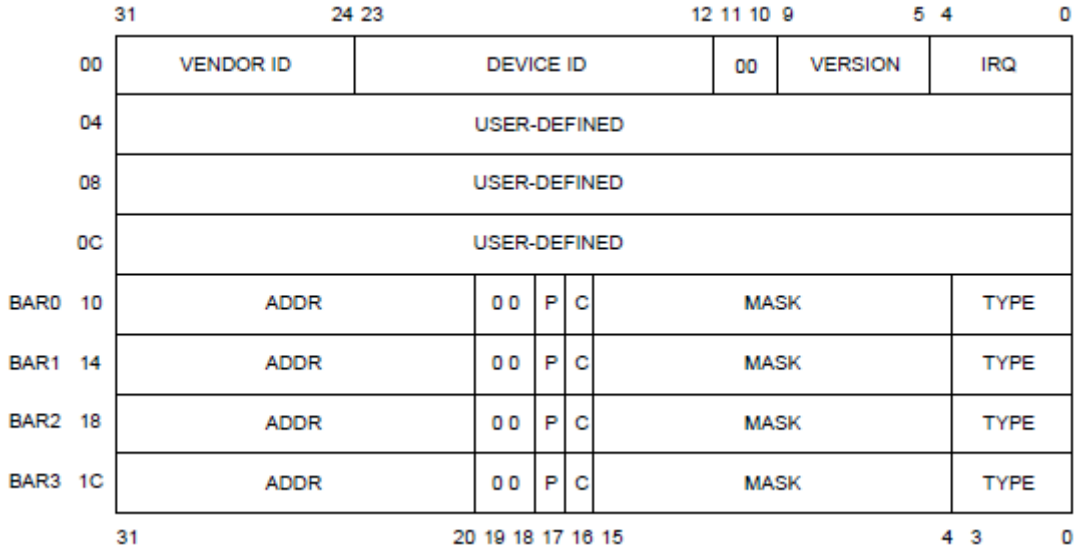
31 24 23 12 11 10 9 5 4 0
00 VENDOR ID DEVICE ID 00 VERSION IRQ
04 USER-DEFINED
08 USER-DEFINED
0C USER-DEFINED
BAR0 10 ADDR 0 0 P C MASK TYPE
BAR1 14 ADDR 0 0 P C MASK TYPE
BAR2 18 ADDR 0 0 P C MASK TYPE
BAR3 1C ADDR 0 0 P C MASK TYPE
31 20 19 18 17 16 15 4 3 0

Figure 4.4: Plug and Play configuration layout for an AHB unit [38]

a given solution.

The reference RASTA board has several interfaces available: Spacewire, MIL-STD-1553, Ethernet, CAN and UART. All those interfaces are based on the VHDL cores developed by Gaisler research and available through the GRLIB library of cores [38]. They connect to the main Leon processor using an AMBA APB or AHB data bus. These buses were introduced in Section 3.2.

4.3.1 AMBA

Accesses to devices mapped on an AMBA bus are transparent to the application. These devices have their interface mapped to the application's memory space, so accesses to regular memory or to a device are similar from an application's perspective.

To ease the development of drivers, the AMBA architecture supplies information about every device attached to an APB or AHB bus in the form of a special table, which is located on a fixed address in memory. The plug and play information for each unit consists of a configuration record containing eight 32-bit words. The first word is called the identification register and contains information on the device type and interrupt routing. The last four words are called bank address registers, and contain address mapping information for the device. The layout of the plug and play table can be seen on Figure 4.4.

The plug and play information for all attached AHB units appears as a read-only table mapped on a fixed address of the AHB, typically at 0xFFFFF000. The configuration records of the AHB masters appear in 0xFFFFF000 - 0xFFFFF800, while the configuration records for the slaves appear in 0xFFFFF800 - 0xFFFFFFFC. This table layout allows for 64 slaves and 64 masters in each AHB bus.

The fields vendor ID, device ID and version identify univocally a given AHB unit. The vendor ID is a unique number assigned to an IP vendor or organization. The device ID is a unique number assigned by a vendor to a specific IP core. Vendor ID 0x00 is used to indicate that no unit is present hence the unit configuration table is empty.

An AHB slave can have up to four address mapping registers (BAR 0-3) and thereby decode four independent areas in the address space. The mask field of the bank address register specifies the size of the memory address region. The Type field specifies the memory address type and finally the P/C flags specify if the memory is cacheable or prefetchable.

The implementation of the APB bus includes the same type of mechanism to provide plug and play support as for the AHB bus. The plug and play information for each APB slave consists of a configuration record containing two 32-bit words. The first word is the identification register and the second the bank address register. Only a single BAR is defined per APB slave. An APB slave memory is neither prefetchable nor cacheable. The layout of the plug and play table for an APB slave can be seen on Figure 4.5.

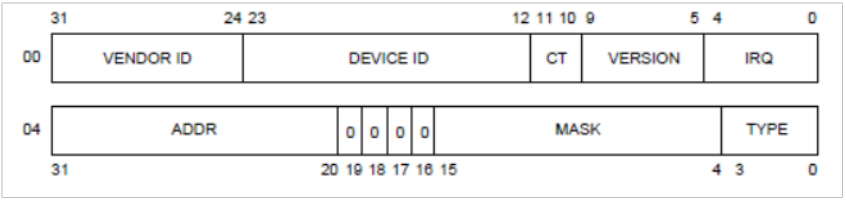

Figure 4.5: APB slave Plug & Play Table. [38]

4.3.2 RS232 - APBUART

The UART – Universal Asynchronous Receiver Transmitter implements serial, RS232, communication. It supports data frames with 8 data bits, one stop bit and one parity bit. Hardware flow control is supported through the RTSN/CTSN handshake signals. The UART cores also features a configurable bitrate and interrupt generation when data is sent or received.

The device is controlled through four memory mapped registers which are displayed in Figure 4.6. For a complete overview of these registers please consult the GRLIB documentation [22].

APB address offset	Register
0x0	UART Data register
0x4	UART Status register
0x8	UART Control register
0xC	UART Scaler register

Figure 4.6: UART registers [22]

Access to the data bus is made by writing a data byte into the UART Data register. This data byte is pushed into an internal hardware, 2 byte deep, FIFO waiting to be sent on the bus. Data reception is similar: when the data register is read, a data byte is popped from the reception FIFO and made available to the application. If one of the FIFOs is full then the last byte is overwritten. On the reception side this can be controlled if the hardware flow control is enabled. On send, the driver must check if the FIFO is full before trying to write a new data byte.

It is clear that this device has very limited buffering capabilities featuring only a two byte deep FIFO. Data must be read as soon as it arrives or it will be overwritten in the next reception.

4.3.3 Spacewire - GRSPW2

The GRSPW2 core implements an interface with a SpaceWire data bus and operates using a direct memory access (DMA) engine which transfers packets directly to and from main memory [38]. The DMA engine interfaces with a Finite State Machine (FSM) through hardware FIFOs. This FSM obeys to the states and behavior defined in the SpaceWire standard (ECSS-E- ST-50-12C [26]). The core is configured through a set of registers accessed through an APB AMBA interface.

Packets with a destination address (first byte of a data packet) corresponding to the address set in the node address register are the only ones accepted by the core. Packets whose address does not match will be discarded silently. If the core's promiscuous mode is active every packet will be dumped in the DMA channels regardless of their destination address.

All packets arriving with a protocol id (second byte of a data packet) different from "0x1" are stored to the DMA channel. Packets with protocol "0x1" (RMAP) are handled by the hardware RMAP core if enabled or stored in the DMA channel if the RMAP core is disabled. RMAP was previously introduced in section 3.2.1.4 as a protocol for remote memory access. When transmitting packets, the address and protocol ID fields must be included in the buffers from where data is fetched. They are not automatically added by the spacewire core.

Direct memory access is based on descriptors located in a consecutive area in memory that hold pointers to buffers where data packets should be stored or read from. These consecutive memory areas make descriptor tables. There are two descriptor tables, one used for transmission and another for reception. The tables have a maximum size of 1024 bytes .

Each table entry is a descriptor and each descriptor is composed by one 32 bit control/status word and one pointer to a data buffer. When a new packet is available, the DMA engine reads a new descriptor. If the descriptor is enabled in the control word, data is written to the data buffer. The size of written data and the status of the operation are written to the control/status word. The core will keep reading the next descriptor every time data is received until it reaches the end of the descriptor table where it wraps to the beginning (descriptor 0).

Write operations are similar to reception: the DMA engine will read the descriptors and, if enabled, write the contents of the data buffer to the SpaceWire bus. An option to generate interrupts on reception or transmission is available on a per descriptor basis.

To avoid buffer overruns the core limits the maximum packet size to a user configurable value. All buffers should be able to cope with a packet of maximum size.

This core has, therefore, buffering capabilities that are directly proportional to the number of descriptors enabled on a given moment. The restrictions on the descriptor table size imply that there is a maximum limit of 64 transmission descriptors and 128 reception descriptors. The core can operate like a terminal, where it only receives data from one spacewire address, or it can act as a monitor that

receives every packet flowing in the network.

4.3.4 MIL-STD-1553B: B1553BRM and GR1553B

The GRLIB library and the RASTA systems provide more than one interface for the MIL-STD-1553B data bus. The B1553BRM core is based on a previous ACTEL core that was adapted to the Leon architecture while GR1553B was developed from scratch by Gaisler Research. Both operate in similar manner and can be in three distinct modes: Remote Terminal (RT), Bus Controller (BC) or Bus Monitor (BM).

Similarly to the other cores already detailed, these devices are configured by a set of registers mapped into I/O memory space and transfer data packets directly into and from main memory. The B1553BRM core is only capable of addressing a contiguous memory are of 128Kb; therefore the size and number of buffers used for DMA is limited.

Communication in a MIL-STD-1553B bus occurs between the Bus Controller and/or Remote Terminals. Each remote terminal occupies one address on the bus and has 31 sub addresses. Sub addresses occupy a similar role to UDP ports and are regularly used for routing inside a remote terminal system. The MIL-STD-1553B standard also defines a special class of commands, named mode codes, that are issued to special sub addresses and which result in special actions like connecting the redundant bus or receiving the onboard time.

B1553BRM

This core is able to operate in one of the three possible operating modes specified by the MIL-STD-1553B standard. It also features the capability of operating simultaneously as a Bus Monitor and as a Remote Terminal. This feature is useful since it enables analysis of the bus traffic.

The core's DMA area must have a maximum of 128 kilobytes of size regardless of the core's operating mode [22]. The memory usage is different for each of the operating modes, but the basic DMA principle is the same and based on descriptors.

When operating in Remote Terminal mode, the allocated memory area is split between a descriptor table containing one descriptor for each subaddress and memory buffers which will contain data. Half of the table is used for transmission while the other half is used for reception. Sub addresses and their respective descriptors are duplicated in the transmission and reception side.

All descriptors have a circular buffer associated, with a configuration defined size. Each descriptor is composed of a control word, pointers to the beginning and end of the circular buffer and a pointer to the current position in the buffer. The core, based on the descriptor control word, will retrieve or place data on the circular buffer and advance the current buffer position. No check is made whether data in the circular buffer is being overwritten. Hence the software must read data from the circular buffers as soon as possible.

Besides writing incoming data for that subaddress in the respective buffer, the core also writes a message information word (MIW) and a time tag: The MIW contains information on the received or transmitted command: word count, mode codes, status info, and any error conditions. The Time Tag

field is set to the value of the internal timer when the MIL-STD-1553B command word has been received and validated. Each MIL-STD-1553B command can carry up to 32 words of data, thus each message on the buffer can have up to 34 words (32 + MIW and Time Tag).

Interrupts can be generated on errors or when data is received and sent. Mode codes can be legalized or illegalized on core configuration. No action will be taken when a mode code command that was illegalized is received [39].

When operating in Bus Monitor Mode, the memory structure is composed of a command block table with up to 1630 command blocks, followed by a data buffer that is used to store command data. The last 32 words are used for logging.

As the bus monitor receives each MIL-STD-1553B message, information regarding the message is stored in a monitor block, an eight-word contiguous block. The core maintains a register that points to the location where the next command block will be stored. When this pointer reaches the end of the command block table it wraps automatically to the beginning of the table. Each block contains a copy of the command word, as well of the status word and of any data sent to the bus.

Bus controller operational mode is different from the other modes. When operating in this mode the core executes a sequence of timed operation in the bus. Operations in the data bus are controlled through the use of command blocks, eight-word, contiguous blocks of memory that contain operation codes for controlling the core as well as MIL-STD-1553B command words and associated data locations in memory.

The core will execute command blocks in a contiguous fashion as long as no “go to,” “branch,” “call,” or “return” operation codes are used. Each command block contains a data pointer to indicate where data for the block is stored. Command Blocks are organized in a table that starts at the memory area base address. Data buffers are placed next to the command block table. The composition of a Bus Controller command block is shown on Figure 4.7.

Word	Function
1	Control Word
2	Command Word 1
3	Command Word 2
4	Data Pointer
5	Status Word 1
6	Status Word 1
7	Branch Address
8	Timer Value

Figure 4.7: Bus Controller Command Block [39]

The command and status words defined in the command block follow the MIL-STD-1553B command specification and will be executed on the bus. The command block control word allows the core to jump to other blocks or to stop command block processing among other operations. A command can be

programmed to generate a CPU interrupt when completed or in error condition.

Each command block has a timer value associated with it. This timer value is the amount of time the core will wait before proceeding to the next command on the table. By specifying a timer value greater than zero it is possible to multiplex the bus access between commands and establish a bus command schedule.

GR1553B

The GR1553B core operates in a similar way to the B1553BRM core but without the memory addressing limitations and with increased functionality. In bus controller mode the core is capable of executing a timed list of MIL-STD-1553B commands autonomously. A second list of commands can be defined and will be executed during bus idle times. For each command in the list it is possible to select the target data bus, the time slot it will occupy and if an interrupt shall be generated on completion.

The Remote Terminal mode uses a three level structure to handle data transfer DMA. The top level is a subaddress table, where each subaddress has a subaddress control word and pointers to a transmit descriptor and a receive descriptor. Each descriptor in turn contains a descriptor control/status word, pointer to a data buffer, and a pointer to a next descriptor, forming a linked list or ring of descriptors. A descriptor is accessed when the core is requested to transmit or received data to a given subaddress. If the descriptor is enabled, data is written or read from the data buffer and the core jumps to the next descriptor. Interrupts can optionally be generated on data send, receive or error.

4.4 Section Summary

This chapter presented the new IMA-SP platform architecture in detail. From the software perspective, the platform is composed by a hypervisor which segregates computing resources, i.e. implements time and space partitioning, and supplies a paravirtualization interface to a partition operating system (POS). The partition operating system provides the multitasking environment inside each partition. In the context of IMA-SP, the selected POS was the space qualified version of RTEMS. On the hardware side, the platform specifies the RASTA development system, with Leon3 and MIL-STD-1553B, SpaceWire, CAN and UART interfaces, as the reference system.

Chapter 5

Input/Output solutions for the IMA-SP architecture

Every kind of computer requires an I/O system in order to perform meaningful operations, and avionics systems are no exception. Without the means to feed data into a system and to get results back there is little sense in performing computations in the first place. I/O, however, is expensive, complex, hardware-dependent and error-prone.

In the last section we obtained a detailed overview of the new IMA-SP reference architecture. The following chapters will overview several possible solutions for the integration of I/O in the IMA-SP partitioned architecture. First we will introduce the difficulties behind the I/O integration in IMA-SP platform, then several hardware and software solutions will be proposed. Finally those solutions will be compared to choose the one that best suits the IMA-SP platform design drivers.

5.1 I/O segregation in partitioned systems

Input/output interfaces are usually hidden by a software layer consisting of operating systems and device drivers. This software layer is usually divided in device independent and device specific (drivers) sub-layers. In embedded systems and real-time applications this is not always appropriate. The time spent on I/O tasks needs to be fully quantified, i.e., it needs to be measured and analyzed like any other part of a real-time system. Even when abstracted by an OS and device driver layer, the I/O timing information, in particular the worst case execution time, must be accessible during design time. Several embedded systems solve this problem by including the I/O code directly in the application, but this increases the application code dependence upon the underlying devices.

In a partitioned system, the quantification of time spent in I/O tasks is even more critical, since it shall be known on whose behalf I/O tasks are performed. The costs for these tasks should be booked to the applications that actually benefit from it. Robust partitioning demands that applications use only those time resources that have been reserved for them during the system design phase. I/O activities shall, hence, be scheduled for periods when the applications that use these specific capabilities are actually

being executed. Furthermore, safety requirements may forbid that some partitions are interrupted by hardware during their guaranteed execution time slices. In consequence, it must be ensured that I/O devices have enough buffering capabilities at their disposal to store data during the time non-interruptible applications are running.

Segregating a data bus is harder than segregating memory and processor resources. I/O handling software must be able to route data from an incoming bus to the application to which that data belongs to. General purpose operating systems use network stacks to route data to different applications. In a virtualized and partitioned architecture, the incoming data has to be shared not only with applications in the same partition but among partitions themselves. Each partition operating system could manage their own devices, but this is only feasible if devices are not shared among partitions. If a device is used by more than one partition then there is the latent risk of one of the partitions leaving the shared device in an unknown state, therefore influencing the second partition. In aeronautical IMA this problem is approached by using partitioning aware data buses like AFDX. AFDX devices are smart in the sense that they can determine to which partition of an end system the data belongs to. This point is critical since the I/O in the platform must be managed in such way that behavior by one partition cannot affect the I/O services received by another

From the rationale exposed in the last paragraphs we can sum up a set of characteristics the I/O system must have to respect the characteristics of a partitioned system:

- The I/O module shall be generic and therefore decoupled from the application;
- The I/O module shall be robust, in the sense that it can be used by more than one partition without interference;
- The I/O module shall be able to route data to its rightful owner (a given application in a given partition);
- The I/O module shall be quantifiable (i.e. its execution time must be bound and measurable);
- The I/O module shall not interrupt, disrupt or have any kind of impact in the time and space partitioning of the applications;

5.2 Integration of I/O into the IMA-SP platform

5.2.1 Software based solutions

5.2.1.1 Integrated in the operating system

The first possible solution for integrating I/O into the IMA-SP platform would be following the classical approach of including the I/O module directly in the operating system. Since I/O must be shared by several partitions this integration would be at hypervisor level, as illustrated in Figure 5.1. The biggest disadvantage brought by this approach is that it would increase the size of the hypervisor. In an IMA system the hypervisor must have a small trusted computing base so it can be easily certified to the

highest levels of criticality. Including the I/O in the hypervisor would increase considerably its size and add complexity to otherwise simple software. An IMA operating system that implements parts of its I/O and networking capability (in particular the TCP/IP and UDP network stack) directly into the kernel is VxWorks 653.

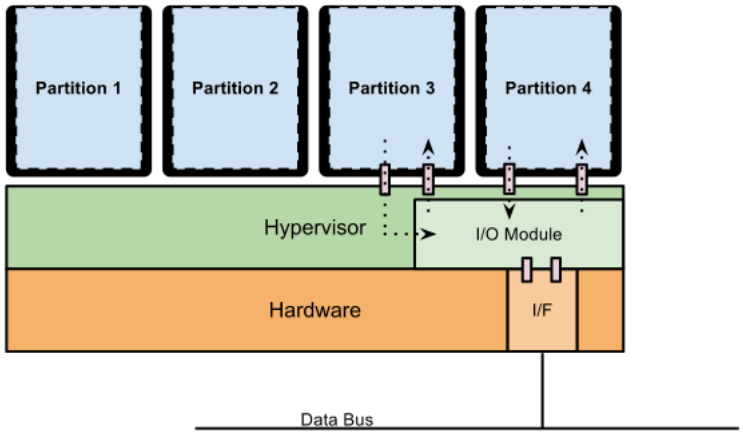

Figure 5.1: Example of an I/O module integrated in the Hypervisor

In a traditional monolithic or modular kernel with I/O integrated, a request to send data from a partitioned application would result in a system call, i.e. a software interrupt. The software interrupt handler would set up a kernel task that processes the sending of data. It would copy the data from the application's address space, and when the device is ready to send data, copy that data into the memory area mapped to the device, or, byte-wise into the device's data registers. The scheduler will take care that the kernel task is pre-empted when either the execution window of the requesting application terminates or another task with higher priority within this application requests the processor.

The scenario for receiving data is generally based on interrupts. The receiving thread will wait until data is available in the device by registering itself as consumer of data coming from this particular device. The kernel, on arrival of this particular interrupt, will re-activate the thread that registered the interrupt and copy the data into its memory space.

When interrupts are disabled for safety reasons, which is common practice during execution of DAL A/B applications, this design is not possible. In such a scenario, the driver has to poll actively for data, and, hence, time has to be allocated for the polling activity on a regular basis.

Note that, when applying this I/O solution to a time partitioned operating system, a policy is needed to schedule I/O tasks. Obviously, on a partition switch, the task must not delay time-critical activities running in the next partition. However, it does not make sense either, to suspend the task until the next execution window of the application on whose behalf it is working. Let's imagine a situation where there are three packets to be sent from different applications. When the driver was writing the first data packet, the associated partition was pre-empted and the driver thread writing the message was suspended. If the scheduler insists on waiting for the next execution window of the sender partition before continuing

with this packet, all applications trying to send data on this device will be penalized with increasing latency. Obviously, this is a problem that could lead to denial of service situations. To avoid this, when the current partition uses the device the processing shall continue with the first packet until it is finished; all applications that use a device have to agree to give some of their processing resources to the I/O subsystem.

5.2.1.2 Integrated in a dedicated partition

In a time and space partitioned environment, the concept of implementing device drivers and network protocols in dedicated partitions suggests itself. Since device drivers require low-level access to hardware, the partition must have the I/O areas or I/O ports mapped into its memory space. In IMA, system partitions that are allowed to use low level services and even to access the hardware directly are often used for such purposes. Figure 5.2 presents a system with a router partition acting as a proxy for all kinds of external communication.

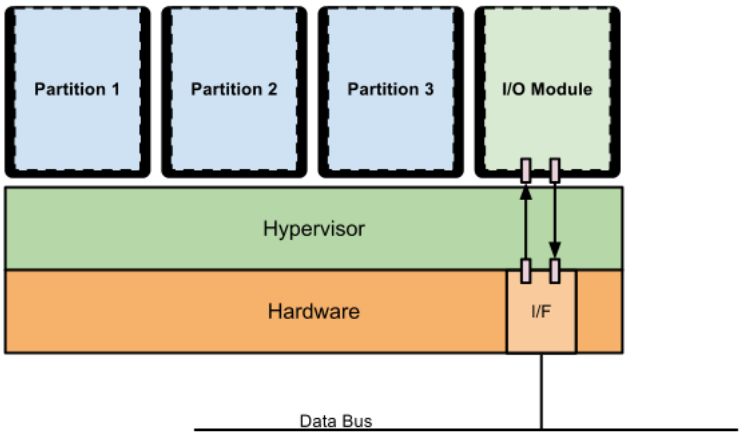

Figure 5.2: Diagram representing the I/O module running in a dedicated partition

The idea is that channels to remote components are routed through the system partition via ports with local channels. This way, a channel connecting a partition with a remote partition or with a sensor/actuator is split into a sub-channel, connecting the partition with the I/O module, and another sub-channel, connecting the I/O module with the remote communication peer.

Since the drivers are encapsulated in a system partition it is possible to add new functionality to the platform in a short amount of time. There is no need for kernel reconfiguration or reimplementation; the system partition brings new functionality, in the same way an application partition does. The approach, hence, strengthens the idea of software reuse and building blocks.

Furthermore, the clear localization of drivers and protocols in a separate memory space eases on-board maintenance: patches can be uploaded to a known memory area, without impacting application or kernel code. It can be even imagined that the I/O partition is unloaded and substituted by a new set of capacities, forming a new mission profile (e.g. safe mode).

The fact that the I/O component is encapsulated in a partition makes hardware interrupts needless; they can be disabled completely. Note that hardware interrupts are not only not necessary, but needless. Even if hardware interrupts were used to signal the arrival of data, there is no means to activate a driver to obtain those data. The device driver will be activated according to the pre-defined partition schedule. The natural way to handle I/O is, thus, polling. Again, even if interrupts were used in this context, the time that is spent on executing the I/O partition is lost, i.e. it cannot be assigned to other activities. For this reason, the time to check for data, when there are none, is not lost.

However, there is also a set of issues related to this architecture. The main problem is latency. The I/O module, being subject to time and space partitioning, does not forward the data directly, instead communication is asynchronous by design. In its execution window, an application issues its data to the I/O module. The I/O module will handle this data only during its next execution window. On the other side of the channel, the same may happen; the I/O module partition receives the data and forwards it to the final destination. But the targeted partition will receive this data only during its next execution window. The path through the network is, thus, extended by the execution windows of I/O modules.

If this principle is applied to a communication scenario where the original sender expects a response from its peer, there will be a delay for all remote communication of at least two router execution windows – one for sending, the other for receiving. This scenario is represented in Figure 5.3.

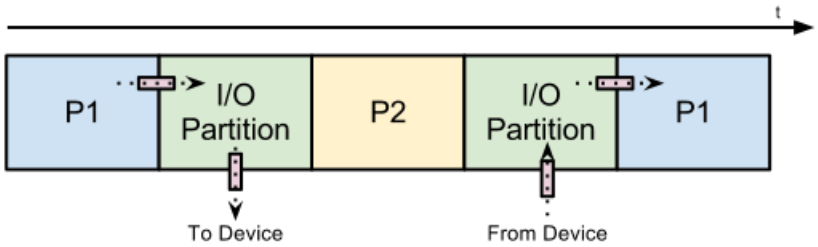

Figure 5.3: Delay in communication originated from the I/O partition being subject to the partition schedule

It may be argued that the execution windows assigned to the I/O module merely reflect the time the system has to spend on I/O and data transportation activity anyway. But in this scenario the cost for the I/O is paid by all parties present in the system. It should be paid only by those applications using a particular I/O in the first place.

Additionally, it must be noted that the approach makes complete execution windows the smallest unit to express time overheads for I/O handling. Execution windows usually have durations of tens or hundreds of milliseconds, whereas the timing requirements for the I/O needs of one partition may often be better expressed in microseconds.

5.2.2 Hardware based solutions

Alternatively to the software-centric solutions, discussed so far, I/O may be handled completely in hardware. The basic principle is to add more intelligence to the interface controllers. We will analyze possible solutions with hardware used today in Space moving forward to more complex solutions like RMAP using SpaceWire. In AFDX, for instance, virtual links are directly assigned memory regions on the receiving or sending computer. Since virtual links are mapped to partitions, a mailbox can be placed inside the address space of a partition. This way, partitions can access the device memory directly, without the risk of leaving the device in an unknown state in case of application failure.

5.2.2.1 MIL-STD-1553B Subaddressing

The MIL-STD-1553B protocol defines thirty subaddresses for each remote terminal device (sub-addresses 0 and 31 are reserved for mode codes). Both MIL-STD-1553B cores available in the RASTA system route data to a specific DMA buffer given the subaddress they are targeted at. Each of these subaddresses is thus a candidate for partition allocation.

MIL-STD-1553B subaddressing presents itself as a possibility to avoid software routing for the incoming packets, since the hardware itself routes each packet to the configured memory area, which can be allocated to different partitions. Regarding transmission, the same principle applies, since there are separate buffers allocated for each subaddress.

By defining at design time the subaddresses allocated to each partition, a configuration artifact can be produced to configure the MIL-STD-1553B device driver (either in the hypervisor or in the I/O partition configuration). The driver would allocate memory for each of the requested subaddresses and map that memory into the destination partition memory space.

Based on this, a very small library could be linked with each partition to read/write data to the MIL-STD-1553B bus that effectively accesses its shared data memory area, without actually interfacing directly with the driver or the hardware except for exchanging control information. The advantage of this approach is that no data needs to be copied from the driver memory area to the application data area (and vice-versa). Data is written directly to the device's DMA memory buffer.

However since there are only available thirty subaddresses, the number of partitions using the MIL-STD-1553B bus is limited also to thirty. This limit in the number of partitions can be lower if the core B1553BRM is used since it can only address up to 128 Kbytes of memory. For each sub-address both read and write operations may be necessary, which demands two separate buffers. Since the memory management unit minimum granularity is 4 Kbytes [21], the actual number of separate sub-addresses mappable to separate partitions is: 128 Kb / (2 ×4 Kb) = 16, minus 1 to account for the control structures, which yields practical maximum of 15 partitions.

It should be noted, also, that a command sent on the MIL-STD-1553B Bus with a wrong sub-address will write into the wrong partition memory area, without detection or without the MMU intervening, since this operation is controlled solely by the IP core.

5.2.2.2 AFDX in space

The basic concepts of AFDX have already been introduced in section 2.3.2. In this section, we will look closer at the integration of AFDX with partitioning, at the concept of Virtual Link (VL). We will also discuss possible adaptations of AFDX to the space domain.

AFDX has a bandwidth of 100Mb/s. The physical link, however, is split into virtual links that allow establishing one-to-many connections similar to the MIL-STD-1553B bus. The AFDX network is configured according to virtual links. Each virtual link is defined with a source and targets and a bandwidth allocation that guarantees data throughput and latency.

The source and targets are mailboxes, which are associated to memory areas. Since the virtual link concept is part of the network protocol, data transported on different virtual links is automatically separated, i.e. it is not necessary to “look” at the packet content to decide who the final target on the receiving computer is. In consequence, the transfer of data from the network link to the target memory area is handled autonomously by the protocol. In IMA, the mailboxes are linked with partitions, such that the memory area to which data is finally copied into is in the address space of an application. Data is, therefore, made available directly to the target application. In the software solutions described before this data routing is a process that needs processing resources and memory to be performed, whereas in AFDX is done automatically.

Note that the AFDX protocol may be implemented by software as well without the use of hardware routing; for instance there are software AFDX solutions from Embvue and SYSGO [40]. Such software implementations, however, usually emulate the advantage of a hardware-based AFDX implementation that copies arriving messages directly into the partition memory by using a second processor core.

Although the delivery of network packets does not need any additional processing power – it is done automatically by hardware (or the AFDX software stack) - there is contention on the bus that may result in a slowdown of application memory access. A system with high data traffic may result in applications with very bad worst case behavior. This must be considered in application design, in particular, if the application does not use the cache or needs to access huge amounts of data or long instruction sequences.

The bandwidth allocation guarantees rely on the subcomponents of the AFDX system, in particular the AFDX cards installed on the end system and the AFDX switches. Cards and switches are configured according to the virtual links. The card, as long as it is working well, will not put more than the configured bandwidth per Virtual Link onto the bus. The switches, additionally, buffer network traffic and emit it according to bandwidth allocation avoiding that one end system “steals” bandwidth from others.

The characteristics supplied by the AFDX protocol are interesting for application in space avionics but as of today there were no attempts to port AFDX into space. The AFDX physical layer is Ethernet. However Ethernet is not suited to use in the space environment because it isn’t radiation tolerant. To port AFDX into space the physical layer needs to be switched to spacewire or another radiation tolerant network. Additionally, end systems and routers should be developed that support a Virtual Link like configuration. Without this router and end systems support, bandwidth and predictability cannot be ensured in the network.

5.2.2.3 Parallel I/O Component

As we have seen in the last chapter there are software AFDX solutions available which make use of extra CPU cores to emulate hardware mailboxes. This approach can be expanded to our scenario and additional CPU cores can be used to host the I/O component outside the IMA schedule. This is, hence, not a pure hardware-based solution. It is nevertheless based on additional hardware means, i.e. a second processor core. A graphical representation of this solution is available on Figure 5.4.

The characteristics of the parallel I/O component are, in consequence, very similar to DMA: the processing is completely separated on a different CPU, so there is no performance loss in processing; there is, however, contention on the bus, leading to wait cycles on both sides, the I/O Network stack and an application running in parallel. This slow-down must be taken into account in application and I/O design.

The objective of the parallel I/O component is to deliver data into the partition address space without the need to schedule additional software on the CPU running the IMA system. The underlying protocol may be SpaceWire or MIL-STD-1553B or even both. The existence of an extra core simply allows detaching the I/O related activities from the IMA schedule.

For receiving, the component either polls periodically through the DMA memory areas or sleeps until interrupted by hardware signaling the arrival of data. On the sending side the situation is similar. It has either to wait for outgoing data, until it is woken up by the OS, or it polls from possible sources of data.

As interface from the application to the I/O component, ports or a shared memory area are used. From the perspective of the I/O component, the difference is not relevant. Even if ports are used, the memory area of the ports can be shared directly with the I/O component. Since the I/O component is at least as critical as the application, there is no risk for the application to share some memory pages with this component. The component, on the other hand, must be robust to avoid being corrupted by a less critical application.

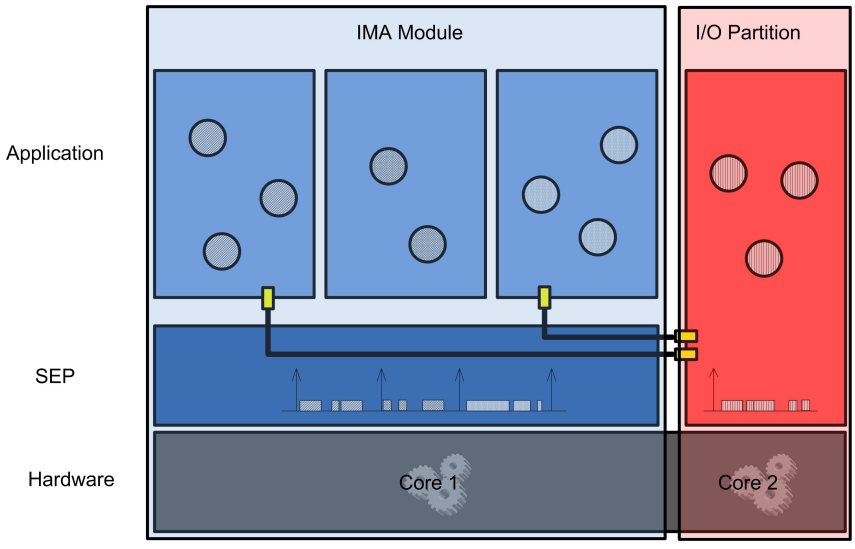

Figure 5.4: Architectural representation of a system with the I/O module detached in a second CPU core [14]

Since a second CPU is used that is independent of the main IMA system, frequent polling does not compete for processing resource with applications on the IMA system and, hence, does not induce latency on procedures implemented there. On the other hand, polling may waste energy by keeping the CPU busy. Note that this is not an absolute statement; the interrupt may also waste energy – or, alternatively, prolong latency in data delivery. When an interrupt is set for each data arrival, the process may get active very often to handle very small amounts of data.

5.2.2.4 Enhancing I/O with RMAP

The RMAP protocol, described in section 3.2.1.4, allows an alternative handling of data packets arriving in the SpaceWire Interface. Data packets that are marked as part of the RMAP protocol id are not handled by the DMA component in the common way. Instead, the RMAP subcomponent takes over. The protocol foresees that data packets can be stored in memory areas defined by the packets themselves.

RMAP, due to its memory access capabilities, can be used to build the I/O component. As pointed out, the bottleneck in any software-based solution is the transfer of data from the DMA memory area into the partition address space. This step is needed, because, in contrast to AFDX, the final destination of a message cannot be determined by the hardware. DMA is able to copy data into different memory areas, but it is not able to detect where a packet belongs to.

With RMAP, this is indeed possible: It is up to the sender to define the memory address the data shall be written to. The sender, hence, needs a local representation of the target memory, at least the areas it is using, to instruct the RMAP device to use this memory according to a policy that is defined on sender side. The sender would need to know the start address of a target buffer, its size and the last address written.

The question arises how the memory area is reused once it has been completely written. The sender may implement a single data area that is always overwritten when new data arrives. Alternatively, the data may be kept in a ring buffer where messages are overwritten only when the buffer is full. An issue that must be solved, however, is the sharing of the ring buffer's control data. This sharing must not introduce a coupling between remote nodes in a network that may result in a joint failure. A possible solution may be to define a time window during which messages shall be kept alive in the buffer; if the time window has expired, the RMAP writer starts to overwrite the oldest messages in the queue without checking for any control data at all. This time out could be enforced by a router in the network, which buffer requests until the timeout expires.

An alternative solution may introduce an acknowledge mechanism on network level. Whenever the ring buffer is full, the sender stops using this buffer until it receives a message acknowledging that the data slots are clear. This protocol introduces some overhead that must be taken into account.

However, the RMAP protocol implementation of the GRSPW core present in the RASTA system opens serious security and safety breaches in the system. This particular core enables remote actors to access memory without being controlled by the memory management unit or the operating system. RMAP, hence, is in this case able to affect the partitioning scheme and even to break the system by overwriting sensible data. Moreover, it is a threat to security policies since it cannot be controlled by the

onboard system itself. Indeed, RMAP bypasses the on-board system and, consequently, passes control to a remote actor. The safety and security of the on-board system cannot be ensured unless by enabling the operating system to control the RMAP transfers.

Since RMAP cannot be controlled by the destination, there is no way to protect the system against arbitrary behavior. In consequence, using such a scheme can only be allowed with some changes in this particular RMAP implementation: It must be possible to restrict the memory areas that may be controlled by RMAP; a packet that tries to access other memory areas shall be dropped. It shall also be possible to restrict RMAP packets according to their sender. When a node has been identified as faulty, it shall be possible to ignore further packets coming from that computer. Otherwise, a remote computer could slow down the target machine by sending RMAP, consuming internal bus bandwidth.

One may argue that turning the protocol around, such that the receiver controls the data exchange by pulling data packets from the sender machine may be a safer solution. But still the solution lacks security standards; even if the RMAP protocol is only allowed to read data, the receiver has access to all memory areas and may “steal” sensitive data.

5.2.2.5 GRSPW2

The GRSPW 2 core has a synthesis option to include more than one DMA channel on the same Spacewire network connection. Each DMA channel will only receive the incoming packets whose spacewire address matches the one on the DMA channel address register. Each SpaceWire address register has a corresponding mask register. Only bits at an index containing a zero in the corresponding mask register are compared. This way a DMA channel can accept a range of addresses. This core, hence, has some hardware routing capabilities.

Mapping the DMA buffers of a given channel to a partition memory space would mean that incoming packets with a given address would be immediately available to the partition, without further software routing. This solution is very similar to AFDX mailboxes, with the role of Virtual Links being occupied by spacewire addresses.

However, since all the DMA channels share the same physical interface with the spacewire bus, contention can be a problem. An application generating an excess of packets in its DMA channel could impact other applications accessing the Spacewire bus through other DMA channels. Moreover, a different DMA channel would need to exist for each partition using the Spacewire bus. Each distinct channel occupies FPGA space; therefore the maximum number of synthesizable channels is limited. With the increase of the number of channels sharing the same physical interface the problems associated with contention also increase.

5.2.3 Comparison of solutions

The solutions previously presented will be compared so the most suitable one is selected for implementation. Table 5.1 lists the main solutions proposed in this document and compares them according to:

	OS solution	I/O partition	Parallel I/O	MIL-STD sub-addressing	RMAP	GRSPW routing
Flexibility	High	High	High	Low	Low	Low
Performance	Medium	Low	High	Medium	High	High
Portability	Medium	High	Medium	Medium	Low	Low
Guarantees	Medium	Medium	Medium	High	High	High
Safety	Medium	High	High	High	Low	Medium
Complexity	Medium	Low	Medium	Low	High	Low

Table 5.1: Comparison of proposed solutions for I/O in the IMA-SP architecture

- Flexibility evaluates if the solution is adaptable/scalable and supports several interfaces (SpW, MIL-STD-1553B, etc);
- Performance is related to possible data throughput and worst case latency;
- Portability is a measure of how independent the solution is from hardware or from a given operating system;
- Guarantees describe the possibility to give concrete guarantees; it describes the solution's capability of being predictable;
- Safety measures the impact of the solution in the overall safety of the system;
- Complexity describes the complexity of the proposed solution;

Some of the solutions described in the previous chapter were not included in this comparison because they are based on hardware that is currently not available, nor planned. The possible values for the evaluation points are: Low, Medium and High.

The trade-off can be based on the axes of achieving much flexibility, high performance, strong guarantees, and good safety levels on one hand against moderate complexity on the other. The OS-based solution appears to be a compromise, since it reaches good results on one hand and only medium complexity on the other. This is also true for the low-level I/O partition which is even "Low" in complexity, but also "Low" in performance (mainly due to restrictions imposed by a static partition schedule).

Hardware based approaches have low values of flexibility and portability since they depend on existing hardware that may not be available in pre-existing platforms. Yet they compensate with high performance, since routing is hardware based, and with low complexity. To be able to develop a flexible and portable I/O solution for today's space reference hardware platform, it must be software based. Solutions depending on the hardware are good candidates for more research activities, because they are able to increase the performance of the I/O subsystem considerably. Nevertheless, the current hardware based solutions are considered to constrain in excess the target system flexibility, i.e. they force the use of specific non-ubiquitous hardware, and are, hence, excluded from selection. Further research activities should explore data buses that have a bandwidth greater than the MIL-STD-1553B bus, while maintaining its determinism and robustness. These data buses should have addressing schemes that support hardware routing.

The solution involving the use of multi core technology obtains the best overall score, however this solution depends on the availability of multi-core processors. Although such systems exist they are still considered on a testing phase. Moreover, the development and qualification of safety critical multi core systems is still a research and development area. Still, multiprocessing support presents itself as a reasonable approach for the improvement of performance of software solutions. This solution can be seen not as a separate entity, but as a natural evolution of the software based solutions. When multiprocessing technology is mature enough for the deployment of safety critical systems, the software solutions presented above can be readily adapted to support it, increasing considerably the system performance. Software I/O modules can be designed to facilitate the multicore transition.

Two solutions remain to be discussed: OS based I/O module and dedicated I/O partition. As seen in the comparison table, the I/O partition has a low performance but a high portability and safety. The I/O partition performance problem is mainly related with the fact that, being a regular partition, it is subject to the IMA partitioning schedule; therefore, a data packet has to wait for an I/O partition execution window before it can be sent, suffering from increased latency. An OS based solution is not so affected by this problem because it is not totally subject to the partitioning schedule. However, an operating system based solution is less portable because it depends, logically, on the underlying operating system. In contrast, an I/O partition can run in any operating system that supplies an IMA-SP compliant interface making it more portable. Other major problem related with the OS based solution is that it increases the trusted computing base of the operating system, making more difficult and costly to qualify or certify it. Another advantage of the I/O partition when compared with the OS solution is that it can be patched or updated during runtime without rebooting the entire operating system. Both solutions can have their performance improved by running in a dedicated core excluded from the IMA schedule.

From the previous reasoning the I/O partition appears as the best I/O solution for today's IMA-SP platform. This solution offers more portability and flexibility while maintaining the complexity on a feasible level. Although it has performance issues, they can be tackled with the inclusion of multi core technologies in the IMA-SP platform. In the future, the research and development of partitioning aware I/O devices should be also able to increase the performance of the I/O subsystem, possibly excluding completely the need for software based router.

5.3 Section summary

This chapter presented, evaluated and compared several I/O solutions for the IMA-SP platform. Both software and hardware based solutions were described. The software based solution that detaches I/O into a dedicated partition was selected due to its high flexibility and low complexity. The main disadvantage of this solution is the poor performance. With the integration of multi-core technologies in the platform, this problem of performance can be mitigated. On the long term, the best option is the development of partitioning aware data buses, which are capable of hardware routing directly into the partition's memory.

Chapter 6

Proposed Solution: I/O Partition Design and Implementation

In the previous chapter several solutions for integrating I/O in the IMA-SP platform have been discussed and compared. The solution based on an I/O partition was selected for development, as the most suited to the requirements. This section will describe the I/O partition development and present a design description and rationale.

As described in section 5.2.1.2, the concept of an I/O partition foresees the encapsulation of all I/O related tasks in one component that is independent from the underlying operating system. The I/O Partition (IOP) must implement all low-level I/O access services foreseen in previous chapters, with special focus on the MIL-STD-1553B, Spacewire and RS232 data buses.

6.1 I/O Component Overview

There are two different scenarios for the usage of I/O resources. I/O interfaces may be shared among partitions or an I/O interface may be dedicated exclusively to one partition. The difference is significant, since in the first case, the access to the I/O interface must be serialized by a trusted component. That the component is trusted means that it has at least the criticality and availability level of the most critical application relying on it. In the second case, no such serialization is necessary. It may be the application's responsibility, i.e. in the scope of the partition of this application, to keep the device in a usable state; no serialization is, thus, necessary.

Independently of the usage scenarios, the I/O component's role is to receive data from applications (one or more), concatenate routing information and send that data to a specific hardware interface. The component must also execute the reverse operation: data must be read from a specific hardware interface and then be routed until it reaches its target application. Consequently, we can divide the I/O module in three logical sub-components: drivers, router and communication middleware. Figure 6.1 depicts the internal architecture of the I/O component.

Drivers implement the access to the hardware; they write and retrieve data from the I/O registers and

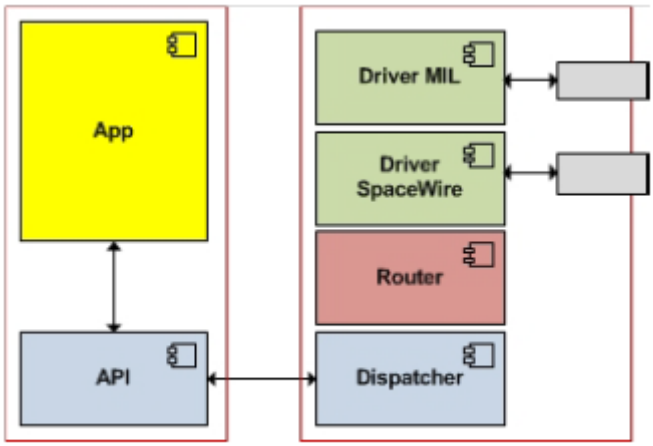

Figure 6.1: I/O component internal architecture

data buffers and change the control registers if necessary. Their main task is to poll for arrived data and pass it to the router or, vice versa, receive data from the router and copy it to the DMA memory of the device. Drivers also abstract the hardware, hence they are specific to a given data bus interface.

The router maps logical addresses to low-level address in terms of the respective protocol (e.g. setting the remote terminal address in MIL-STD-1553B or the destination address in SpaceWire) and vice versa. Its main role is to determine to which device data shall be sent and to which partition the data belongs, by using the protocols specific to the underlying data bus.

Finally, the communication middleware is responsible for the communication with application partitions on the same computer. Data received through a hardware interface is delivered through innate communication means of the IMA-SP platform, e.g. ARINC 653-like queuing and sampling ports or segments of shared memory. Applications use those communication ports in an identical way to deliver data to the component.

In the case where a hardware interface device is used by one partition exclusively, no data has to be exchanged with a proxy component. Instead, the I/O memory used by the device can be mapped directly into the address space of the application. This is shown in Figure 6.2. The internal architecture of the I/O component remains similar with just some adjustments in the interface between the application and the component.

This design is optimal in terms of latency; the data will be available to the application or for sending, respectively, as soon as it has “arrived”, i.e., as soon as the hardware has finished its task. It is also an advantage that no additional resources are needed to exchange data between consumer/producer applications and I/O interfaces.

Care, however, has to be taken in this approach with the use of DMA. The DMA component of a device, dedicated to one application, is working on behalf of this application – and only this application – and shall, consequently, count as activity of this partition. The DMA is, however, not scheduled by the operating system and, hence, won’t be stopped by the OS like other activities running in the current partition. The partition is not trusted and as such cannot be responsible for controlling DMA. If the control

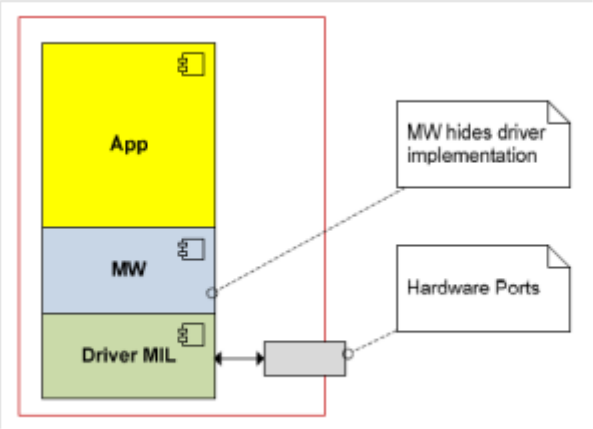

Figure 6.2: I/O component architecture when dedicated to a single partition

is not implemented in the OS DMA activities have to be taken into account in the run-time analysis of applications. It must be assumed that for each application, there may be concurrent access to the system bus that is not related to the activities of this application.

Until now, the partitioned I/O component has been seen as a unique partition in the system. This, however, is a special case of multiple I/O partitions. Indeed, as other application components, sub-components of the I/O component may be implemented in different partitions. There are several potential advantages of such a design.

In terms of scheduling, it is possible to separate I/O drivers. An application that only uses a subset of drivers requires only that the partition with the respective drivers is activated before or after its own execution. The scheduling of the separated partitions can, hence, be optimized according to the needs of consumer applications. This, however, can also be achieved by assigning appropriate periods on process level. With respect to scheduling, multiple I/O partitions may help separating I/O components, but is not strictly necessary.

Separating submodules in different partitions is, nevertheless, a good means of separating concerns. This is in particular an interesting feature in terms of qualification. When only a subset of less critical applications need a certain I/O capability, the related driver needs to be qualified only at the level of these applications; more critical drivers will be restricted to a smaller subset of very critical applications. There may be typical I/O profiles for tasks with different criticalities, such that a given device is typically used by less critical applications. In this case, a qualification at low criticality level is sufficient. This may be an interesting perspective for bringing a component from a low to a high criticality level, by applying it first in less critical context.

6.2 I/O Partition Design

The I/O partition can be implemented on different levels of abstraction: It may act as a pure driver in which it handles raw data that is copied to the device interface or it may also provide access to high level

services such as protocols, platform specific services or file storage. These objectives are very different. Whereas the first approach aims much more at the implementation of general purpose components for a generic operating system, the second approach implements service-specific components. The layering, however, also implies dependencies. The second kind of services will require some kind of the first set of services.

Since service specific components are considered to be outside the scope of this thesis, the I/O partition will define only low-level I/O access services. This means that higher level services or on-board functions will use the defined API to access these I/O services and implement the needed higher level protocols. For instance, a telemetry middleware or telecommand management unit will use the I/O component, but the component itself will have no knowledge of telemetry or telecommands. The same applies to other kind of services, all built on top of this component.

The I/O partition shall provide generic interfaces to control devices, send data via a data bus or to receive data from a data bus. These services will be abstracted and accessed by the communication middleware. User applications send requests for these services via inter-partition communication means (i.e. Shared memory or ARINC 653 queuing and sampling ports).

Starting from I/O interfaces in general purpose operating system, one can conclude that there are four generic interfaces that the component shall implement to provide low level access to applications:

- **Read** - The read service requests data from a given service which may be an on-board device or a component on a network;
- **Subscribe** - The subscribe requests reads on periodically or sporadically created data. The subscribe service identifies an abstract topic related to an application component or a device which periodically or sporadically creates data. The I/O partition, when receiving such data, sends them back to the subscriber;
- **Write** - The write service requests sending data to another application or to a device; the standard mode for the write service is “fire-and-forget”; an alternative mode that foresees an acknowledgement from the target component may be useful;
- **Control** - The control service is not primarily intended to send data to another application, but to change control settings of a device; it may be used to disable or enable certain properties of a device or to disable/enable the device at all;

When using the I/O module, an application has only to provide data and a device identifier to request an I/O access. The I/O partition is responsible for handling all specific routing involved in the I/O access. From the application’s perspective, I/O is reduced to accessing a set of identifiers named logical devices. Inside the I/O module, a logical device is translated into routing information that is appended to data. Hence, the topology and addressing in each data bus is abstracted by the I/O module. As an example, a developer can create a logical device named MAGNETOMETER that is mapped, by the I/O partition, to the Spacewire 0 bus in the address 125.

Due to redundancy reasons, a device may be accessible in more than one data bus simultaneously. Consequently, issuing a request to a logical device may result in accesses to more than one bus. For

instance, the GYRO device could be in IP 10.0.0.255 in bus ETH0, but could also be in Spacewire1 address 250. In IOP terminology GYRO is a logical device because GYRO is the logical representation of one or more bus positions/addresses related to the same device/service.

6.3 I/O Partition architecture

6.3.1 Interfaces

The IMA-SP platform defines a set of standard interfaces between operating system kernel and applications; one of these interfaces is the space-tailored and qualified version of RTEMS [29], another is a subset of the ARINC 653 APEX [34] tailored to the needs of space applications. The I/O component uses RTEMS primitives for internal task management and ARINC 653 ports to communicate with applications. There is additionally the option to use shared memory to optimize the handling of huge amounts of data when the underlying operating system supports such a feature.

From an application's perspective there are three ways to access the I/O partition services: through the middleware API, remote ports or shared memory. All these interfaces are controlled by a subcomponent called the communication middleware. This communication middleware is divided into two subcomponents. One is part of the I/O component itself (dispatcher), the other is linked with the application and serves as library providing an API and configuration support (middleware).

For most cases, the underlying communication mechanism will be a set of queuing ports. This set of queuing ports is used to send requests to the I/O component and to send results back to the sender if the current mode is not "fire-and-forget". In some cases where huge amounts of data are to be exchanged an additional interface shall be used, such as shared memory. This interface shall be configuration controlled.

When an application requests one of the above mentioned services, it calls a function implemented in the middleware. This function accepts the data to be sent and respective size as parameters, adds a header according to the service and sends this as a queuing message to the I/O component; alternatively, the data will be copied into a shared memory segment and the I/O partition will only be informed by a short message pointing to the segment.

The middleware (MW) library defines an API which provides access to the I/O Partition services. Using the MW any partition can require read, write or control operations on a given device or data bus. The Interface between the MW and the I/O partition is implemented by two queuing ports, one in each direction or by a shared memory segment. There are four primitives available in the MW:

- **io_init** - Initializes internal memory structures and queuing ports. This function needs to be executed during initialization and before any other MW function;

- **send_read_request** - This primitive is used to send a read request to the I/O partition, so that any available data from the specified device is sent to this partition. The RID parameter is used to later obtain the requested data using the get_reply primitive;

- **get_reply** - This primitive obtains a reply to a previous request. This reply can contain data, if the request identified by rid was a read request, or just the status of the operation, if it was a write request;
- **send_data** - This primitive requests a write operation to a target logical device. The success of this operation can be acknowledged by the IO partition. If this ACK was requested the get_reply service shall be used to obtain it using the rid value returned by this call;

Internally, the middleware encapsulates the parameters and data onto a memory structure similar to the one defined bellow. This structure contains fields for specifying the desired operation and target device as well as for the data and data size.

```
typedef struct service_request{
        int device;                 /**< target logical device */
        unsigned int request_id;    /**< id that identifies this request */
        operation_enum operation;   /**< 0-Read; 1-Write; xx - others */
        unsigned int ssize;         /**< efective data size */
        char generic_data[MAX_BUFFER_SIZE];
} service_request_t;
```

Additionally to the middleware interface, the I/O partition is also able to link ARINC ports with devices via configuration. This, in compliance with ARINC 653, is transparent to applications. A remote ARINC 653 port is linked with a local I/O partition, instead of with a local application partition; data written to this port by the application is automatically sent, by the I/O partition, to a network interface defined by configuration means. The data is read on the other side by a counter I/O component and written to an ARINC 653 port that, eventually, links with the final target application. Each I/O partition involved needs to have a mapping between ports and addresses/devices so it can forward data correctly.

With the remote port system a partition doesn't need to be aware that it is communicating with a remote device. The partition communicates as if it was communicating with another local partition, but instead is sending/receiving data from a remote device. Data sent to a configuration defined ARINC 653 port will be forwarded automatically to the logical device associated with it. The inverse operation is also possible: data originating from a given logical device is automatically sent to a given port.

6.3.2 Internal Design

Internally, the I/O component consists of a set of tasks that are periodically executed according to the partition frequency and execution time. Its internal schedule, i.e., the sequence of tasks execution according to their periods and priorities, is devised in order to always achieve a full write and read cycle per partition execution window. This schedule is depicted in Figure 6.3.

The first tasks to run in a execution windows is the dispatcher; it is responsible for obtaining data from the queuing ports and from shared memory segments, and to append that data into the corresponding logical device. The target logical device from a given data set is determined differently depending on the

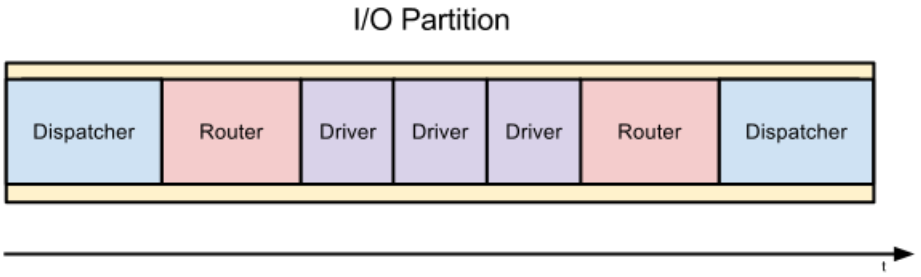

Figure 6.3: I/O partition internal task schedule

external I/O partition interface being used. If the interface used is the communication middleware API then the target logical device is passed as part of the API parameters. For the remote port system, each port used has logical device mapped to it.

Once the dispatcher has finished, all requests and data will be distributed through their respective logical devices. Internally, a logical device is just a memory structure that maps a logical device identifier (e.g. IMU) to a set of routes. These logical routes will do the mapping between the logical device and a given address on a given data bus. As an example, the route for the logical device IMU is defined next (IMU is on the MIL-STD-1553B bus 0, address 15, subaddress 10).

```
static logical_routing_t imu_routes[] = {
        {
                .active = 1,
                .dev = BRM0,
                {
                        .milstd_header = {
                                .desc = 15,
                                .address = 10
                        }
                }
        }
};
```

Any operation requests or incoming data is appended to the logical device structure as a node in its doubled linked lists. Doubled linked lists were chosen because they are very simple and an API for handling them is available in the operating system (RTEMS). Moreover, for the set of operations required, append and remove, the algorithmical complexity of linked lists is O(1) and, therefore, optimal. The memory structure defining a logical device can be found bellow.

```
typedef struct logical_devices_s {
        int route_number;               /**<Number of Routes*/
        logical_routing_t *lroute;      /**<Logical-> Physical routing*/
        Chain_Control sendqueue;        /**<Data to be sent*/
```

```
        Chain_Control pending_rcvqueue; /**<for read requests*/
        Chain_Control rcvqueue;         /**<received data for this logical*/
} logical_device_t;
```

The second task to run is the router. It will remove write requests pending on each of the logical devices and, if required, append the routing headers to data buffers. Also, it will determine to which of the physical devices the request will be routed to. A physical device is a memory structure representing a physical data bus (e.g. spacewire 0). A driver exists for each one of the physical devices. After adding the routing headers, the router task appends the requests to a linked list in a physical device structure. The memory structure defining a physical device can be found bellow with its respective linked lists.

```
typedef struct physical_devices_s {
        int route_number;            /**< Number of Routes */
        physical_routing_t *proute; /**< Physical -> Logical routing */
        Chain_Control rcvqueue;      /**< Data that was from this device */
        Chain_Control sendqueue;     /**< Write requests */
} physical_device_t;
```

Once the router has finished distributing all requests to the physical device structures, it is time for the drivers from each physical device to run. Drivers fetch requests from their physical device structure and write the respective data and routing headers to the data bus (by DMA or polling). An I/O control operation may not involve writing data, but changing some control register of the device. This act terminates a write cycle and corresponds to a half complete I/O cycle. A graphical representation of the write cycle is presented on Figure 6.4. After completing all write requests, the driver starts the second half of the cycle by obtaining any incoming data from the data bus. This data will be appended to another linked list in the physical device structure.

Subsequently, the router runs once again and fetches the previously read data. The routing information in the data buffers is parsed and extracted according to its specific bus dependent format. Each physical device has a set of routes to which any incoming data is compared: if the header in the incoming data is equal to the one on the route, then the incoming data is appended to the logical device the route points to.

When the previous step is completed, all incoming data packets will be appended to their respective logical devices and the dispatcher will start to run once again. Now, the dispatcher role is to verify if there are pending requests for each one of the logical devices or if they are mapped to an ARINC 653 port by configuration. If any of these conditions is true, the dispatcher will fetch the data and sent it through the respective port or memory mapped segment to its rightful owner. This completes a full I/O cycle (read and write).

To better understand the role of each one of the subcomponents in the I/O cycle, it is better to look at a concrete I/O access example. Let's imagine that an attitude control application wants to send data to an on-board actuator, for example a reaction wheel. Physically, the reaction wheel is attached to the SpaceWire bus on the address 128. As part of the I/O partition configuration, the developer needs

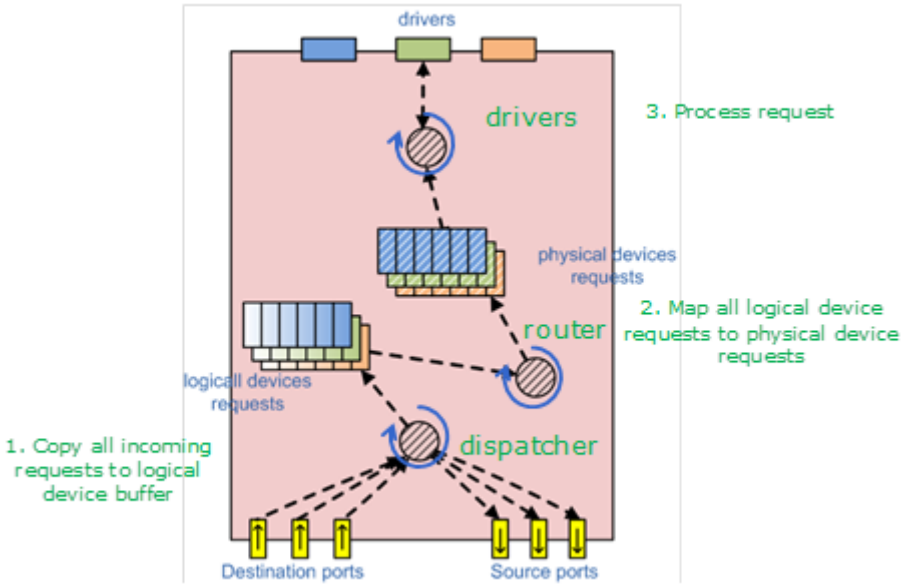

Figure 6.4: IOP internal I/O Cycle

to create a logical device to represent the reaction wheel and associate a route with it. This route will map the logical device identifier chosen (e.g. “RW”) with the address 128 in the SpaceWire bus. If the developer wishes, he can also map, by configuration, a queuing or sampling port directly with the “RW” logical device.

When the I/O partition runs, the dispatcher will read from every queuing port, sampling port and shared memory segment. Some of the ports will be associated with the middleware, while others with the remote ports system. When reading the port linked to the “RW” logical device, the dispatcher will find some data sent by our fictitious application. It’s the dispatcher role to encapsulate this data and to append it to the RW logical device structure. If the application developer used the middleware API, the data would have appeared in the queuing port associated with middleware. In this case, the target logical device identifier would have been passed along with the data in the queuing message (it’s a parameter in the MW API).

Once all data requests are appended in their respective logical devices, it is time for the router to run. The router cycles through all logical devices and routes their data requests. When it reaches the RW logical device structure, it will find the data sent by our application. The router’s role is to create a routing header in the data request and append this request to the respective physical device structure. In this specific case, the router will insert the reaction wheel’s SpaceWire address in the first position of the data buffer. The data buffer will now have the format: {128, data[0], . . . , data[n] }, as specified in the SpaceWire protocol. After the completion of this step, the router must append the processed data buffer to the SpaceWire physical device structure. The router stops processing when all requests are routed to their respective physical devices.

The next tasks to get executed are the several device drivers. The SpaceWire driver will obtain all requests from its physical device structure and write them to the SpaceWire bus. This driver is a dumb

in the sense that has no knowledge of routing; it only writes a data packet. All the routing was taken care by the router task . Once all write requests are written to the bus, the driver checks for incoming data and appends it to its physical device structure . This starts the read cycle.

6.4 Configuration

The configuration process of the component relies on an independent tool chain. The configuration defines:

- The logical addressing used by applications (logical devices and respective routes);
- The mapping to the physical addressing used with the I/O drivers (physical devices and respective routes);
- The selection of a set of device drivers;
- Device configuration parameters, such as the use of DMA, DMA memory areas, etc;
- Inter-partition communication ports or shared memory segments;

A tool chain fulfilling those needs is represented in Figure 6.5.

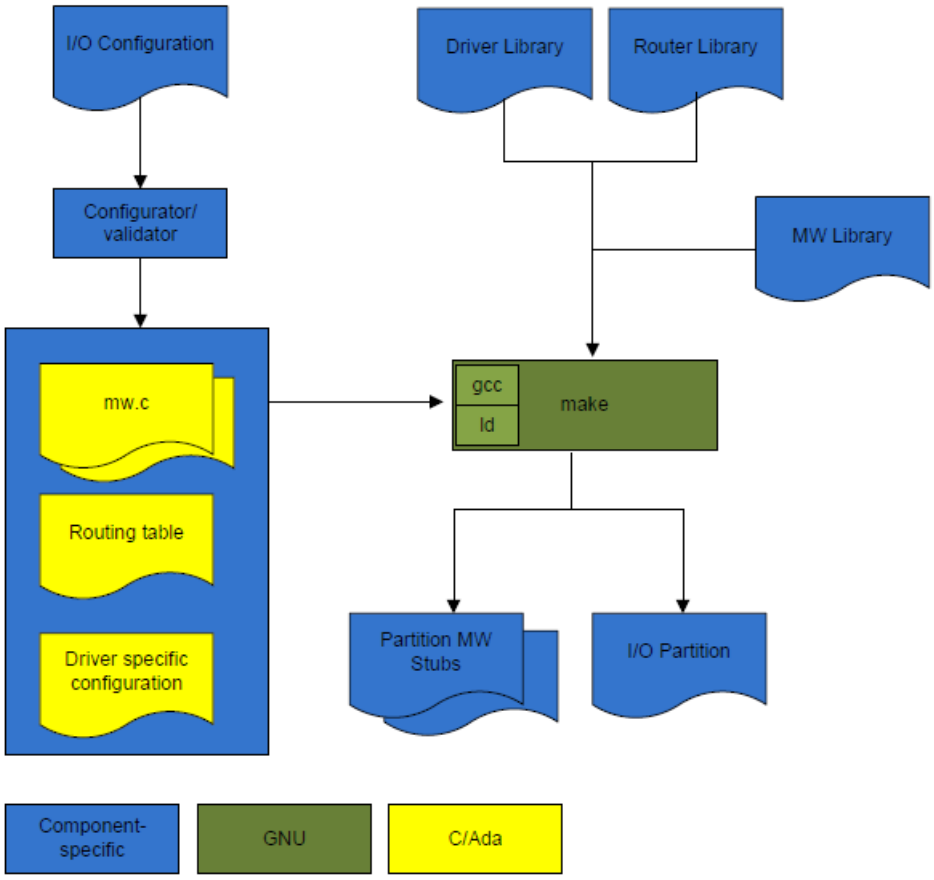

Figure 6.5: I/O Configuration toolchain

The tool chain is entered through the configuration artifact, e.g. an XML file, in the upper left corner of the figure. This artifact is processed by a specific tool that produces a set of machine-readable objects encoded in Ada or C; the first type of objects is the middleware stubs. Each of these stubs is dedicated to one application and is, hence, injected into the OS specific tool chain to build this application. The MW Library, as well, has to be linked with the application code and thus to be injected into the application tool chain.

The middleware stubs encapsulate the inter-partition communication. Note that this may imply duplication of some configuration data that are needed by the I/O component and by the OS, in particular communication ports. The duplication may be overcome by generating OS-specific configuration from the I/O tool chain and injecting it into the OS tool chain.

The next object is the routing information that maps logical addressing to protocol-specific addressing. Finally, there are driver-specific configuration data. These objects are compiled and linked with the I/O partition code, in the case of shared devices; in the case of a dedicated I/O device, the driver specific configuration will be linked, like the MW stub, with an application. An application partition may, hence, contain up to four I/O-related objects: the MW stub, the MW library, the router library and the driver library.

More details about the I/O partition configuration can be obtained in Appendix A – I/O partition user manual.

6.5 The MIL-STD-1553B challenge

A particular configuration problem is presented by the MIL-STD-1553B-interface. MIL-STD-1553B is not just a channel for data transportation, but a configurable system to control remote components. A MIL-STD-1553B bus controller operates by executing a predefined set of bus commands during strict time windows. These commands are usually grouped in structures called minor frames, which are in their turn grouped in a major frame. A major frame acts as a fixed duration schedule of MIL-STD-1553B commands that is repeated in time with a given frequency. The MIL-STD-1553B command schedule is synchronized with the application scheduling so that data is timely available to the commands and applications being executed [41]. This synchronization is depicted in Figure 6.6.

The synchronization between application and bus controller schedule poses a bigger challenge in a time partitioned operating system because the applications are bound to a two level schedule. This problem can become complex because both the partition schedule and the application schedule inside each partition must be altered to ensure synchronization with MIL-STD-1553B. Additionally, the complexity can increase if the bus controller command list must be exclusively managed by an I/O partition. In this case, in order to feed data to the command list the I/O partition needs to run immediately following a user application.

Even if the MIL-STD-1553B and application schedules are synchronized by master frames, a drift between the two can built up over time because they are clocked from different sources. To avoid this drift some systems create synchronization points using a real time interrupt [41]. Both schedules start

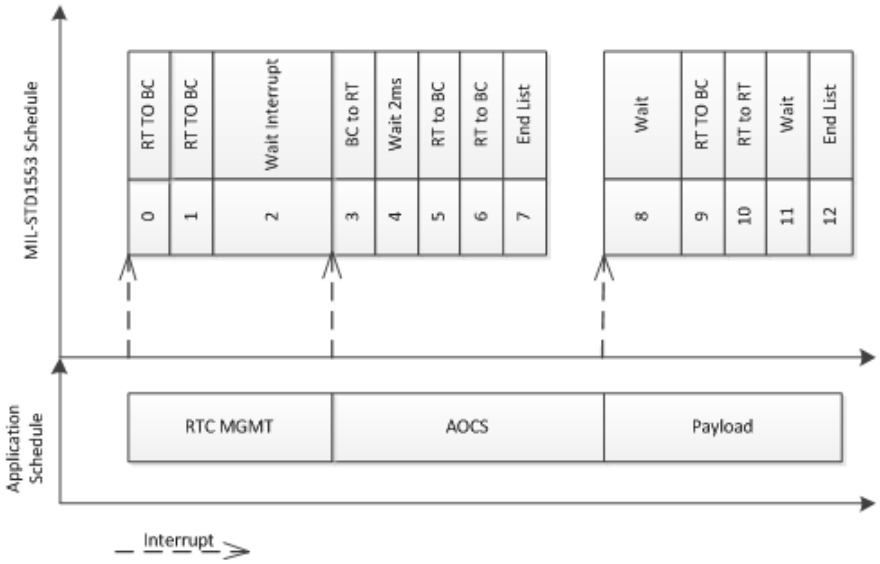

Figure 6.6: Example MIL-STD-1553B and application schedule synchronization

simultaneously at each synchronization point, therefore ensuring that no drift exists. In satellite on board systems these interrupts are provided by a real time on board timer that ticks with a given frequency. Synchronization with an external interrupt means that a partition window would have to wait for the interrupt, therefore extending or reducing its duration. This type of synchronization creates a problem to time and space partitioned operating systems since the partition scheduling is, according to the ARINC 653 specification, static (i.e. all partitions have well defined and strictly limited durations).

6.5.1 Proposed solution

In the I/O partition, the bus controller command lists are statically defined as part of the configuration process. Multiple command lists can be configured; these lists can be switched during run time to provide support to different system operating modes. Depending on the underlying target device these commands can not only be MIL-STD-1553B bus commands but also control flow commands like branch or terminate list.

To each command on a given list a Slot Id number is attributed (Figure 6.7). This Slot Id is used across the configuration and API to refer to this specific command. The configuration also allows mapping a given Slot Id, and therefore its respective command, with a local ARINC 653 port. Data written to this port by the application will be transported to the correspondent MIL-STD-1553B command. The Slot Id is also used to identify data buffers when the underlying MIL-STD-1553B device has shared memory support.

If the used MIL-STD-1553B core has no specific memory constrains, thus being able to access the full memory address range, it is possible to use shared memory exposing only the data buffers used in the MIL-STD-1553B commands to the applications. In this case the I/O partition assumes uniquely the role of managing the device's state and not the data transfers. The mapping between a given command and the partition that has access to it must be done at the configuration level. All buffers belonging to

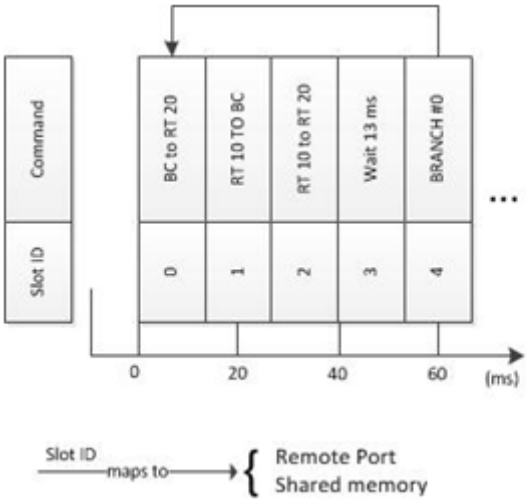

Figure 6.7: A MIL-STD-1553B command list

a partition are bundled together and mapped to the partition address space. Since the data buffers are accessible by the application there is no requirement to run the I/O partition to feed data to the command list; this role is done directly by the application. A special user side API is used to manage these data buffers, ensuring that the correct data buffer for a given Slot Id is written or read.

For MIL-STD-1553B devices where it is not possible to map the data buffers in the application address space, the I/O partition has to be responsible for handling data input/output on the command list. Instead of writing directly to the data buffers, the application will send data to the I/O Partition, which writes it to the command list during its own execution windows. The synchronization between the bus and application will be more complex because the I/O partition has to run each time a bus data access is required. In this situation the I/O partition must also track the completed commands and forward any incoming data to the respective partition. Each command may be accessed through a request/reply API, or by direct mapping, by configuration, to a sampling port.

Nevertheless, the above described solution doesn't address the requirement of schedule synchronization. In non partitioned systems, this synchronization is achieved by means of an external interrupt that synchronizes the MIL-STD-1553B and application schedules.

To solve this problem it is proposed to insert a buffer partition at the end of the partitioning schedule. This partition has a well-defined window start time but no fixed duration. Its real duration will be constrained by an interrupt originating from the MIL-STD-1553B core. The core must be programed to generate an interrupt at the very end of its MTF. When the buffer partition catches this interrupt, it can trigger a partition context switch, therefore starting a new MTF at application level.

To increase the robustness of this solution, a MIL-STD-1553B interrupt that happens outside the Buffer partition's window will raise a fault event (i.e. start the health monitor). Additionally, a maximum execution time can be attributed to the buffer partition. If the MIL-STD-1553B interrupt does not occur until the end of the execution time an FDIR event is raised. The interrupt that signals the MTF end must also reset the timer counter that drives the regular clock tick for the operating system. This ensures that the first window of the next MTF has the correct duration. If the timer is not reset, the next window could

lose up to 1 tick of execution time (when the MIL-STD-1553B interrupt occurs in the last instant before the clock tick interrupt).

A graphical representation of the solution is shown below. The buffer partition's window is exacerbated to show the partition context switch occurring on the MIL-STD-1553B interrupt. On a normal case, the drift between the last clock tick and the MIL-STD-1553B interrupt should be very small (<< one clock tick).

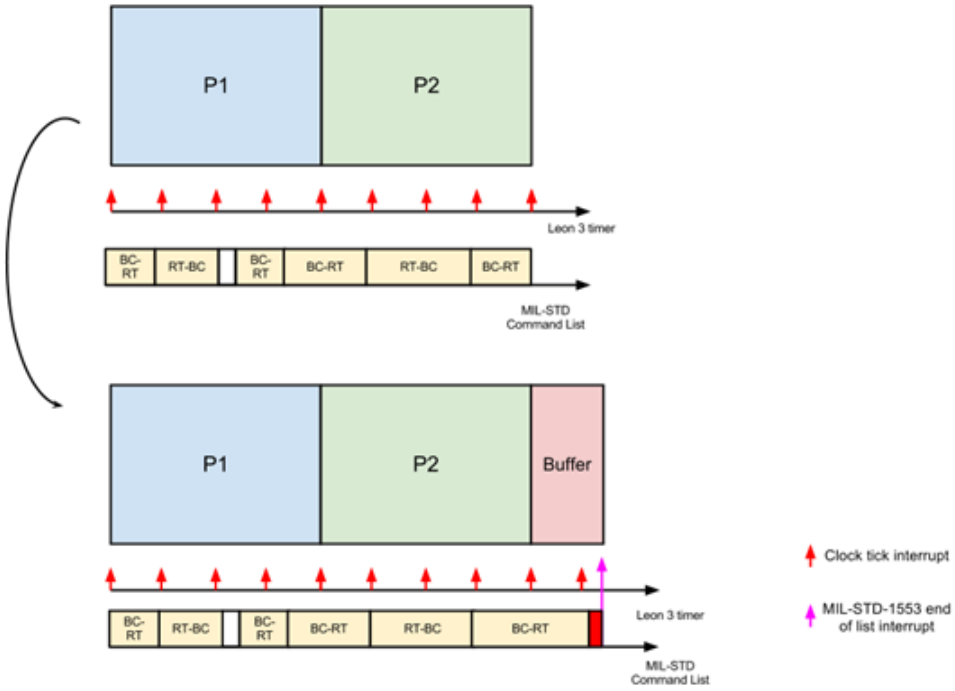

Figure 6.8: Concept of Elastic Partitions

Chapter 7

I/O Partition Testing

In the last chapter, the I/O partition internal design was presented. Some problems faced during the design and implementation of the MIL-STD-1553B subsystem were also introduced and one possible solution was described. In this chapter, the preliminary testing and benchmark activities related with the I/O partition will be presented.

7.1 Functional Testing

To demonstrate the functionality and capabilities of the I/O partition several demonstrators were developed; each one for a different underlying data bus. The application layer of the demonstrators remains constant, and the only change is the I/O partition configuration that is specific to the data bus being used.

These demonstrators are based on a control system. The control system is modeled after a typical control application, consisting of a plant (sensor and actuator), a controller and a Human Machine Interface (HMI) showing the current value of the controlled entity. The value that is controlled by the application is nothing particular. It is a simple integer value. You may think of something like engine temperature, cabin pressure or ground speed.

The three components of the system are hosted on different on-board computers. The plant, used in the example, implements logic to change the controlled value in the same way an environment would do (i.e. the plant component simulates the system dynamics). The most important factor influencing the value of the measured quantity is the state of the actuator. The actuator may be in one of three states: Incrementing, Decreasing or None. The HMI is just a simple graphical interface showing the measured value.

The purpose of the control application is to keep the controlled value within a given range by changing the state of the actuator and to update the representation of the value in the HMI. To achieve this, the control application communicates with the plant and the panel. From the plant, it receives the current sensor value and forwards it to the panel whenever it has changed; if necessary, it changes the state of the actuator by sending an actuator command to the plant.

IOP logical device name	ARINC 653 remote port name	Source address	Destination address	Comment
"ACTUATOR"	“ACT”	192.168.0.128:6667	192.168.0.1:6667	Actuator data from the control to the plant
"HMI"	“HMI”	192.168.0.128:1911	192.168.0.1:1911	HMI status data from the control to the HMI
“SENSOR”	“SENSOR”	192.168.0.1:6666	192.168.0.128:6666	Sample data from plant that is sent periodically to the control.
SIMSERVER	N/A (uses MW)	192.168.0.128:13000	192.168.0.1:13000	Loopback send
SIMSERVER	N/A (uses MW)	192.168.0.1:13000	192.168.0.128:13000	Loopback receive

Table 7.1: Communication channels used in the Ethernet demonstrator.

In the I/O partition demonstrators, the control component is running in the AIR operating system with one I/O partition. This system is then hosted on a RASTA Leon3 computer with relevant I/O interfaces. The plant and HMI components are hosted in general purpose computers running Linux. The Linux computers are equipped with I/O interface cards for the data buses used in the demonstrators.

In addition to the control application example, there are two partitions in the onboard computer communicating using a remote echo destination. Partition 1 sends data to a remote destination which replies the same data back to the module. This data is then forwarded to Partition 2.

From a software perspective, the only change between the various demonstrators is the data buses used. There are demonstrators for the following data buses: Ethernet, Spacewire and MIL-STD-1553B. Only the Ethernet demonstrator will be detailed in this text since it uses a regular UDP/IP address scheme. The SpaceWire and MIL-STD-1553B demonstrators have the same interfaces as the Ethernet demonstrator but the communication between the development board and the Linux computer is done through a SpaceWire or MIL-STD-1553B Bus. The communication channels remain the same except that they are mapped on top of a SpaceWire or MIL-STD-1553B bus.

The Ethernet demonstrator assumes that the development board is connected to the Linux machine using a crossover Ethernet cable. During the testing, the computer running plant was assigned with the IP address 192.168.0.1, while the development board got the IP address 192.168.0.128 (Defined in the IOP configuration). The communication channels listed on Table 7.1 are defined in the I/O partition configuration to support the control-plant communication and the remote loopback.

The control application running in an AIR partition receives sensor data from plant through the “SENSOR” queuing port. This queuing port is connected with the I/O partition. Due to its configuration the I/O partition will write any data incoming from 192.168.0.1:6666 (plant’s IP) to that queuing port. To send actuator commands to the plant and status data to the HMI, the control application writes data to the “ACT” and “HMI” queuing ports, respectively. These queuing ports are mapped by the I/O partition to

the addresses 192.168.0.1:6667 and 192.168.0.1:1911 in GRETH0.

The partition remote loopback communication doesn't use the remote port system; instead it uses the middleware to access the I/O partitions services. Partition 1 sends a write request for the "SIMSERVER" logical device while the partition 2 makes a read request for "SIMSERVER" data.

7.2 Preliminary Benchmarks

To create representative benchmarks the typical usage profiles have to be defined. The MIL-STD-1553B commands list master frames in space avionics systems generally have frequencies in the order of dozens of Hz, with several synchronization interrupts occurring in a master frame. Any realistic use case for the I/O component should be able to cope with timing requirements in this order of magnitude. The major issue with this data bus lies in schedule synchronization and not in data throughput.

Whereas the MIL-STD-1553B bus is used for communicating with low bandwidth devices with strict timing constrains, the SpaceWire bus is generally used for data transfers between high bandwidth applications and devices with soft real time requirements. A suitable use case for this data bus must test the data throughput of the IMA-SP module under stress conditions.

An example configuration was devised with two partitions, each communicating with a remote terminal on the MIL-STD-1553B bus. The I/O partition is executed during 2 ms at each 6 ms, thus with a frequency of 166 Hz. The first two milliseconds of the frame are occupied by an application that sends data to a remote terminal, while the last two milliseconds are used by an application that receives data from the same remote terminal.

The MIL-STD-1553B command master frame is composed by a RT to BC transfer occupying the first 4 ms and by a BC to RT transfer using the remaining 2ms. The synchronization between the application and MIL-STD- 1553B schedules is depicted in the Figure 7.1.

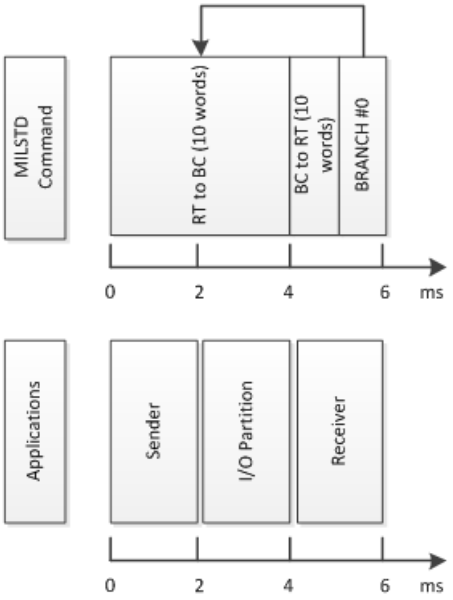

Figure 7.1: MIL-STD-1553B benchmark

The MIL-STD-1553B schedule was built to ensure that the data is timely available to the applications and to the MIL-STD-1553B command. No external synchronization methods were used, therefore synchronization only relies on the matching Master Frames. We were able to send and receive the MIL-STD-1553B messages on each task, achieving synchronization during the 24 hours of test execution.

To further test the command definition interface, the underlying MIL-STD-1553B core was switched and the example was executed again successfully; Therefore demonstrating that the I/O partition's MIL-STD-1553B microcode is really core agnostic.

The SpaceWire benchmark consists of a system composed by one application partition and one I/O partition. Each partition is allowed to run for 5 ms with a frequency of 100 Hz. The application partition tries to send 10 megabytes of data in loopback through the Spacewire bus using messages of 1 kilobyte. This test yielded a system data output of about 2 megabits per second. This value can be explained by the bottleneck generated by the memory copy operations in the interface between the two partitions. The usage of shared memory, bigger messages and a I/O partition optimized for SpaceWire could easily increase several times this value.

Chapter 8

Conclusion

This chapter presents an overview of the topics discussed in this thesis and of the major conclusions obtained.

8.1 Overview

The research work presented on this dissertation focused on the development of a module providing I/O capabilities for the Integrated Modular Avionics for Space platform. This novel software architecture for space avionic systems leverages upon the success of its aeronautical counterpart to become the natural successor of the current executive. However, since the underlying hardware platform is inherently different for space avionics, the input/output solutions used in the aeronautical systems are not currently applicable to it.

Several possible architectures for the I/O module, derived from the current space and aeronautical platforms, were proposed and compared. It was concluded, in Chapter 5, that the most fit solution foresees the encapsulation of all I/O related tasks in one dedicated system partition. This specific solution is the one that offers more portability and robustness while maintaining a low level of complexity; it is also very adaptable in the sense that it can evolve as the platform matures.

The main difficulty related with the I/O partition concept is that it is scheduled like a normal partition, introducing high latency in communications. Maintaining latency at controlled levels while respecting other applications and system scheduling constraints may prove to be an impossible task. This problem must be evaluated on a per case basis. Nevertheless, latency and scheduling difficulties apply, to some degree, to all the purely software based solutions presented. The problem itself stems from one of the key points of this architecture, time partitioning, and is, therefore, very hard to handle without specific hardware support.

Besides the latency problem, it was found that the configuration the I/O module is one of the biggest challenges faced. As part of an on-board system, the I/O partition shall provide real time guarantees and a well defined and bound execution time. This execution time is directly related with the amount of data that has to be exchanged in a given execution window. Obtaining a worst case execution time estimation

for an I/O partition interacting with multiple application partitions and with multiple hardware devices is a very complex task. Furthermore, the estimation must be done on a per execution window basis since each execution window can have different data throughput requirements (in terms of hardware devices and amount of data).This task shall be undertaken by the system integrator since it may involve inputs from several application suppliers.

Another issue to consider is the impact of the direct memory access channels on the worst case execution time of the applications. Even though the I/O partition is limited to run in its assigned execution windows, the DMA transfers are directly initiated by the hardware device and not limited to any window. An application accessing main memory will have to compete with the memory accesses of the DMA channels, resulting in delays. This issue must be tackled by the introduction of a representative I/O data access profile during the application's WCET estimation. This specific problem does not originate from the I/O solution chosen, but from the hardware platform. It can only be fully solved by careful hardware design.

8.2 Dissemination

A conference paper entitled "An I/O building block for the IMA space reference architecture" [42] was produced based on the results of this dissertation. This paper was accepted and presented on the International Space System Engineering Conference - DASIA 2012 - that was held in Dubrovnik, Croatia in May 2012.

8.3 Current State

The following features of the I/O partition architecture, described in Chapter 6, were implemented and tested in the scope of this thesis:

- WRITE, READ, IOCTL and SUBSCRIBE interfaces;
- Remote ARINC 653 ports;
- The dispatcher/router and subscription environment;
- A RS-232 driver (APBUART);
- A deterministic IP stack providing UDP on top of an Ethernet drive (GRETH);
- SpaceWire drivers (GRSPW and GRSPW2);
- MIL-STD-1553B drivers for Bus Controller and Remote Terminal modes (AS1553b and BRM1553b);
- MIL-STD-1553B core independent microcode to define MIL-STD-1553B commands;

8.4 Future Work

The MIL-STD-1553B approach that uses shared memory and schedule synchronization was not completed in this scope because it depends on operating system features that are not available at the moment. This improvement is left as future work. Notwithstanding this missing feature, the I/O partition is fully functional and able to interface MIL-STD-1553B buses through the middleware API or the remote ports system.

Another possible improvement, left as future work, is the development of support tools that ease the I/O partition configuration process. Currently, this configuration is done by directly changing code structures that are linked with the I/O partition. The creation of a graphical user interface that automatically generates the necessary code stubs would decrease significantly the efforts required for the configuration process. The configuration tool could also follow a ARINC 653 like approach and use xml based configuration files.

An important point that was considered outside of the scope of this dissertation is the development of a comprehensive test suite for the I/O module. As part of a critical system, the I/O partition needs to be fully qualified; this qualification process requires a complete test suite composed of structural (integration, unit, validation) and functional tests.

The natural evolution path of the I/O partition is through the exploitation of parallel computer architectures. This can be accomplished in the short term because space domain specific processors are currently migrating to multi-core configurations. The usage of a dedicated core for I/O processing would result in a considerable improvement in terms of data throughput and latency. As all the I/O tasks are encapsulated in one single partition, it would be fairly easy to deploy the I/O partition on a dedicated processor core. Nonetheless, the parallelization of the I/O component depends on the availability of multi-core aware operating systems.

An alternative approach to parallelization would be the employment of specific operating system features that enable the occupation of scheduled but unused time. Real time systems have operating margins in which the processor is idle; these margins correspond to the difference between the worst case execution time of a given task and its actual execution time. Some operating systems provide mechanisms to schedule tasks in this otherwise unused time. One of these mechanisms is the Co-Partitions feature available in the AIR operating system [43]. Co-partitions are able to run in the unused time of an application execution window, without compromising the real time characteristics of the system. Deploying the I/O partition as a Co-Partition would allow to reduce the latency and improve performance.

On the long run, the best solution for the platform is the development of new partition aware data buses, possibly following the AFDX model. This hardware development solution may replace entirely the software based I/O module. These hardware devices expose the data buffers and some control registers directly to the partition's memory space. No additional CPU resources, or increased latency, would be needed for I/O in such a configuration. For the time being, however, a software module is still necessary to maintain robustness and get the job done.

Bibliography

[1] Henning Butz. Open integrated modular avionic (IMA): State of the art and future development road map at Airbus deutschland. Technical report, Department of Avionic Systems at Airbus Deutschland GmbH, 2010.

[2] European Space Research and Technology Centre, Keplerlaan 1 - 2201 AZ, Noordwijk - The Netherlands. *Statement of Work: Integrated Modular Avionics for Space*, 1 revision 4 edition, September 2009. Reference: TEC-SWS/09-281/SoW.

[3] European Space Research and Technology Centre, Keplerlaan 1 - 2201 AZ, Noordwijk - The Netherlands. *Avionics Time and Space Partitioning User Needs*, 1 edition, August 2009. TEC-SW/09-247/JW.

[4] Jean-Bernard. A380 integrated modular avionics: The history, objectives and challenges of the deployment of IMA on a380. Technical report, AIRBUS S.A.S., 2010.

[5] Christopher B. Watkins and Randy Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *26th Digital Avionics Systems Conference*, Grand Rapids, Michigan, 2007.

[6] José Rufino, João Craveiro, Tobias Schoofs, Cássia Tatibana, and James Windsor. AIR technology: A step towards IMA in space. In *Proceedings of the Data Systems in Aerospace Conference (DASIA 2009),*, Naples, Italy, May 2009.

[7] James Windsor and Kjeld Hjortnaes. Time and space partitioning in spacecraft avionics. In *Third IEEE International Conference on Space Mission Challenges for Information Technology*, 2009.

[8] Radio Technical Commission for Aeronautics (RTCA). DO-178B: Software considerations in airborne systems and equipment certification, 1982.

[9] AERONAUTICAL RADIO, INC., 2551 RIVA ROAD, ANNAPOLIS, MARYLAND 21401-7435. *AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE PART 1: REQUIRED SERVICES*, 3 edition, November 2010. ARINC SPECIFICATION 653P1-3.

[10] AERONAUTICAL RADIO, INC., 2551 RIVA ROAD, ANNAPOLIS, MARYLAND 21401-7435. *AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE PART 2: EXTENDED SERVICES*, 1 edition, December 2008. ARINC SPECIFICATION 653P2-1.

[11] Paul J. Prisaznuk and AEEC staff. ARINC 653 role in integrated modular avionics. In *27th Digital Avionics Systems Conference*, St. Paul, Minnesota, 2008.

[12] José Rufino, Sergio Filipe, M. Coutinho, Sérgio Santos, and James Windsor. ARINC 653 interface in RTEMS. In *Proceedings of the Data Systems in Aerospace Conference (DASIA 2007)*, Istanbul, Turkey, May 2007.

[13] João Pedro Gonçalves Crespo Craveiro. Integration of generic operating systems in partitioned architectures. Master's thesis, Universidade De Lisboa Faculdade de Ciências Departamento de Informática, 2009.

[14] Tobias Schoofs and João Cristovão. I/O handling strategies. Reference: Integrated Modular Avionics for Space/D06; Contract ESTEC 4000100764, April 2011.

[15] AIM GmbH, Sasbacher Str. 2 79111 Freiburg, Germany. *ARINC 429 Specification Tutorial*, 2.1 edition, November 2010.

[16] TechSAT GmbH, Poing. *AFDX/ ARINC 664 Tutorial*, 1000 edition, August 2008.

[17] G.R.P. Francois. Study on I/O in time and space partitioned systems. Technical Report NLR-CR-2008-531-PT-1, National Aerospace Laboratory NLR, November 2008.

[18] K. Hjortnaes and J. Windsor. System requirements and architecture for time and space partitioning in spacecraft. In *Spacecraft Flight Software (FSW-10)*, 2010.

[19] European Space Agency. Microelectronics technologies for space. `http://www.esa.int/TEC/Microelectronics/SEMSTCV681F_2.html`, October 2012.

[20] Aeroflex Gaisler AB, Kungsgatan 12, Göteborg, Sweden. *LEON3FT-RTAX Data Sheet and User Manual*, 1 edition, October 2009.

[21] SunSoft, A Sun Microsystems, Inc. Business, 2550 Garcia Avenue, Mountain View, CA 94043 U.S.A. *SPARC Assembly Language - Reference Manual*, 1994.

[22] Aeroflex Gaisler AB, Kungsgatan 12, Göteborg, Sweden. *GRLIB IP Core User Manual*, 1.0.22 edition, April 2010.

[23] Dallas Semiconductor. *Application Note 83 Fundamentals of RS232 Serial Communications*, 1 edition, 1998.

[24] National Instruments Corporation. *Controller Area Network (CAN) Tutorial*, 1 edition, November 2011.

[25] AIM GmbH, Sasbacher Str. 2 79111 Freiburg, Germany. *MIL-STD-1553b tutorial*, 2.3 edition, November 2010.

[26] European Cooperation for Space Standardization (ECSS). ECSS-E-ST-50-13C; space engineering: SpaceWire - links, nodes, routers and networks, March 2008.

[27] European Cooperation for Space Standardization (ECSS). ECSS-E-ST-50-52C; space engineering: Spacewire - remote memory access protocol, March 2010.

[28] OAR Corporation. RTEMS real time operating system. `http://www.rtems.org/`, October 2012.

[29] Helder Silva, José Sousa, Daniel Freitas, Sergio Faustino, Alexandre Constantino, and Manuel Coutinho. RTEMS improvement space qualification of RTEMS executive. In *INForum2009*, Lisboa, Portugal, 2009.

[30] Marie-Hélène DeRedempt. IMA-SP system executive platform: Software system specification. Reference: IMA-SP/IMA SP SEP SSS; Contract ESTEC 4000100764, March 2012.

[31] Marie-Hélène Deredempt, Alain Rossignol, James Windsor, Joaquim Sanmarti Regis De-Ferluc, Jason Thorn, Paul Parisis, Fernand Quartier, Francis Vatrinet, Tobias Schoofs, Alfons Crespo, Julien Galizzi, and Paul Arberet. Integrated modular avionics for spacecraft software architecture and requirements. In *Proceedings of the Data Systems in Aerospace Conference (DASIA 2012)*, Croatia, May 2012.

[32] Luc Planche. Requirement baseline: Top level requirements;. Reference: Integrated Modular Avionics for Space/D01; Contract ESTEC 4000100764, December 2010.

[33] A. Crespo, I. Ripoll, M. Masmano, P. Arberet, and J.J. Metge. XtratuM: an open source hypervisor for TSP embedded systems in aerospace. In *Proceedings of the Data Systems in Aerospace Conference (DASIA 2009)*, Istanbul, Turkey, May 2009.

[34] Régis de Ferluc. TSP services specification. Reference: Integrated Modular Avionics for Space/D10; Contract ESTEC 4000100764, May 2012.

[35] Sandi Habinc, Jiri Gaisler, Claudio Monteleone, and Chris Taylor. Critical spacewire elements in RASTA. In *Proceedings of the Spacewire Conference 2008*, Nara, 2008.

[36] Aeroflex Gaisler AB, Kungsgatan 12, Göteborg, Sweden. *GR-RASTA Product Sheet*, April 2012.

[37] Aeroflex Gaisler AB, Kungsgatan 12, Göteborg, Sweden. *GR-RASTA User Manual*, April 2012.

[38] Aeroflex Gaisler AB, Kungsgatan 12, Göteborg, Sweden. *GRLIB IP Library User Manual*, 1.1.0 b4100 edition, October 2010.

[39] Actel Corporation, Mountain View, CA 94043. *Core1553BRM v3.1 Handbook*, 3.1 edition, 2010.

[40] P. Khazali, I. Boulais, and M. Cole. AFDX software network stack implementation -practical lessons learned. In *Proceedings of the Digital Avionics Systems Conference, DASC 2009)*, Orlando, FL, 2009.

[41] European Cooperation for Space Standardization (ECSS). ECSS-E-50-13C; space engineering: Interface and communication protocol for mil-std-1553b data bus onboard spacecraft, July 2008.

[42] Cláudio Silva, João Cristovão, and Tobias Schoofs. An I/O building block for the IMA for space reference architecture. In *Proceedings of the Data Systems in Aerospace Conference (DASIA 2012)*, Dubrovnik, Croatia, May 2012.

[43] Tobias Schoofs and João Cristovão. Sharing the costs of common tasks in the AIR partitioning operating system. In *Proceedings of the Data Systems in Aerospace Conference (DASIA 2011)*, San Anton, Malta, May 2011.

Appendix A

I/O Partition User Manual

A.1 I/O Partition Concepts

The I/O Partition is a component that supplies access to remote devices to local applications. Several data buses can be interfaced by the IOP: MIL-STD-1553B, SpaceWire, UART and Ethernet. Applications issue requests to the IOP which will, in turn, communicate with the target device to send or receive data on behalf of the application. The IOP, in spite of its name, can be either linked directly with applications or hosted on a separate partition. The asynchronous communication, however, is the same in both cases.

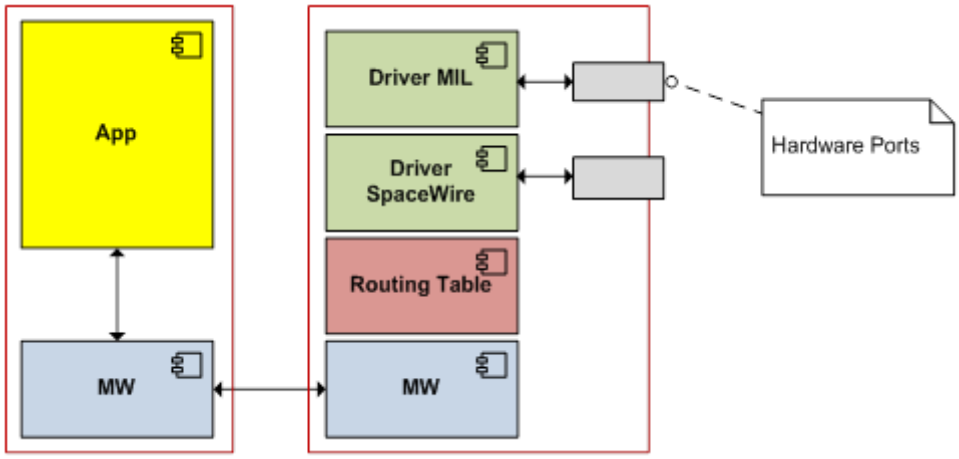

Figure A.1: Application partition communicating with I/O partition.

The IOP supplies logical level access to remote devices. This fact means that the application has only to handle data, leaving any logical level routing to the I/O partition. As part of the IOP configuration the developer defines the logical positions where a given remote device can be reached. The I/O partition will then route incoming data to those devices as requested. User applications only operate by issuing requests on these logical identifiers.

For instance, the GYRO device could be in IP 10.0.0.255 in bus ETH0, but could also be in SPW0 address 250. In IOP terminology GYRO is a logical device because GYRO is the logical representation of one or more bus positions/addresses related to the same device/service.

Each logical device defined in the IOP configuration will have a set of assigned routes. These logical routes will do the mapping between the logical device and a given address on a given bus. The opposite is also valid; each physical bus will have a set of routes to which any incoming data is compared. If the header in the incoming data is equal to the one on the route, then the incoming data is assigned to the logical device the route points to.

The services provided by the I/O partition can be accessed from the user point of view through two distinct systems: the middleware and the remote ports systems. These interfaces are described in detail in Section A.2.5 – I/O Partition Interfaces.

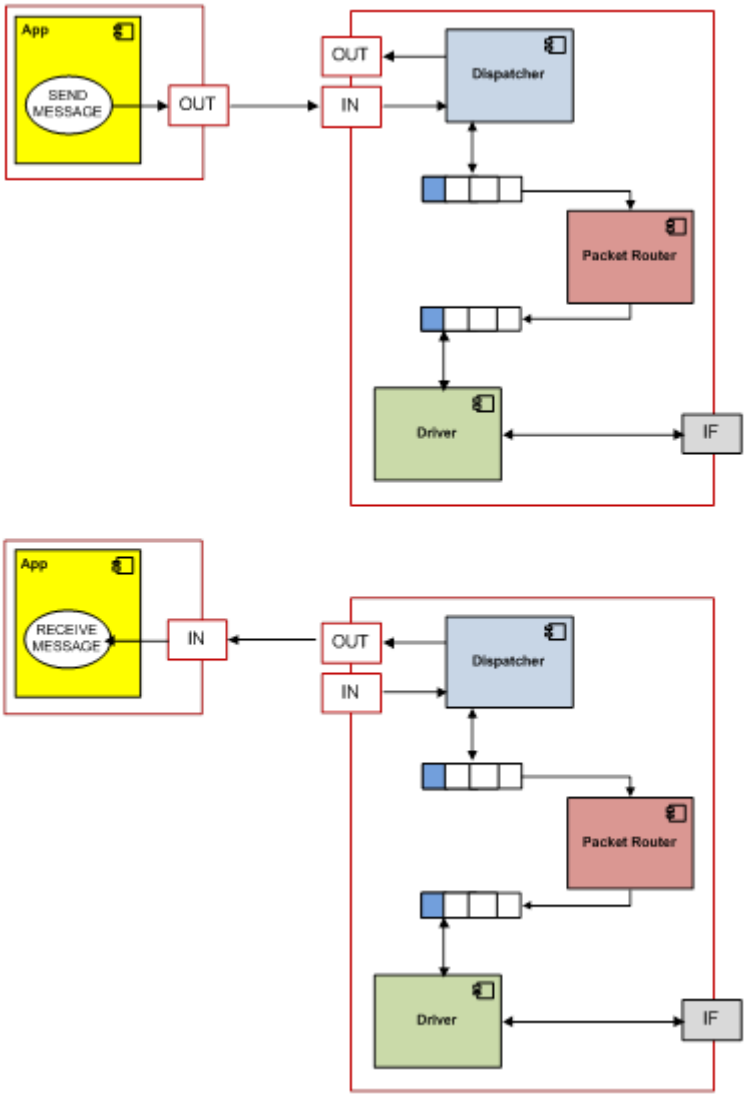

Figure A.2: Interface with I/O partition through remote ports.

A.2 I/O Partition Configuration

The configuration of the I/O partition can be split in different sub configurations: generic, ports, routes and device specific configurations.

A.2.1 Generic Configuration

The first step in the configuration is to select which data busses the partition must interface with. As these options affect the building process, the I/O partition library must be recompiled every time they are changed. The available data buses are defined in the file *IOPconfig.h*. The following configuration example uses three SpaceWire devices and two MIL-STD-1553B Bus controllers:

```
#define GRSPW
#define GR1553B
#define GR1553BC
typedef enum {SPW0, SPW1, SPW3, BRM0, BRM1} physical_dev_enum;
```

Next you must identify devices and their corresponding position on each bus. This identification process will yield the logical devices you need to configure on the I/O partition and their respective routes. A logical device is just a name that represents a given address on a given bus. The logical devices are defined using a *device_enum* type:

```
typedef enum {STDOUT, STDIN, STT, GYRO, ACTUATOR, HMI} device_enum;

/** Number of Logical Devices */
#define LOGICAL_DEVS 6
```

There are other generic configuration options of less importance: *iopschedule[]* defines and IOP period in ticks and *phy_dev_reads_per_period* indicates how many data packets may be read from the Hardware in one partition execution window for each data bus. This last array is indexed by the *physical_device_enum*. Finally, *timeto_live* value represents the maximum number of partition periods that a request should remain unhandled before being discarded.

A.2.2 Routing Configuration

Each logical device needs to have a route associated with it because the I/O partition needs to know how to address that device. A route establishes the bridge between the logical device and a given address on a given data bus. A route is only followed if it is marked as active. This enables alternate routes to be activated with different operating modes.

The route specifies the physical data bus where the device is present and an address field that is relative to that bus. A logical device can have more than one active route at a time. This can be useful in situations where a device is in hot redundancy and therefore has to be accessed in more than one bus.

Logical routes are defined by filling a structure of type *logical_routing_t*, whose members are defined in Table A.1.

This structure defines if the route is active, the target data bus and an address header. The address header depends on the target data bus since each bus requires a different addressing scheme.

In order to increase the flexibility of the IOP routing configuration, logical routes are only used to route data originating from the I/O partition. For data that is received, physical routes are used. The

Type/Name	Possible Values	Description
int **active**	integer [0-1]	Is this route active?
physical_dev_enum **dev**	SPWn, BRMn, ETHn GRBn	Databus to where this route is aimed at
milstd_header_t **milstd_hdr**	valid milstd header	Defines a MIL-STD-1553B header comprising a descriptor and a subaddress
spw_header_t **spw_header**	valid SpW header	Defines a SpaceWire header comprising a SpaceWire address and its size
unsigned char **udpiphdr[42]**	N/A	Only used during runtime
eth_header_t **eth_route**	valid Ethernet header	Ethernet Header formed by MAC address, IP address and source/destination UDP ports

Table A.1: *logical_routing_t* structure members used to define a route

principle behind them is the same. They are used to route incoming data to a logical device using the address.

Physical routes are associated to each data bus interfaced by the I/O partition. When new data arrives, the header on the incoming data is compared to each physical route of that data bus. If there is a match between the two, the data is forwarded to the logical device defined on the physical route.

Physical routes are defined by filling a structure of type *physical_routing_t*. Its members are defined in Table A.2.

Type/Name	Possible Values	Description
int **active**	integer [0-1]	Is this route active?
device_enum **dev**	logical device identifier	logical device where this route is aimed at
milstd_header_t **milstd_hdr**	valid milstd header	Defines a MIL-STD-1553B header comprising a descriptor and a subaddress
spw_header_t **spw_header**	valid SpW header	Defines a SpaceWire header comprising a SpaceWire address and its size
unsigned char **udpiphdr[42]**	N/A	Only used during runtime
eth_header_t **eth_route**	valid Ethernet header	Ethernet Header formed by MAC address, IP address and source/destination UDP ports
int **subsribe**	integer [0-1]	Don't append data to the logical device, but send it directly to the defined ARINC 653 port
int **sampling_port_id**	integer [0-REMOTE_PORTS]	Queuing port to where the data shall be sent.

Table A.2: *physical_routing_t* structure members used to define a route.

This structure defines if the route is active, the target logical device and an address header. The address header depends on the physical bus to where this route is attached since each bus requires a different addressing scheme.

Each route shall only have the header referring to its own data bus defined. (e.g *spw_header* for spacewire routes). The header contents for the SpaceWire and MIL-STD-1553B buses are defined

next:

```
typedef struct spw_header_s {
        unsigned int hlen;      /**< Header Length */
        unsigned char hdr[32];  /**< Header */
} spw_header_t;

typedef struct milstd_header{
        unsigned char desc;     /**< Subaddress */
        unsigned char address;  /**< RT Address */
} milstd_header_t;
```

A.2.3 Routing Configuration Example

Below is shown the example of a logical device named ACTUATOR that is available on the bus Spacewire0 with the address 100. This example uses SpaceWire path addressing; therefore the first byte of the address is the SpW router port to which the actuator device is connected (port 3 in this example).

```
static logical_routing_t actuator_route[] = {
        {
                .active = 1,
                .dev = SPW0,
                {
                        .spw_header = {
                                .hlen = 2,
                                .hdr[0] = 3,
                                .hdr[1] = 100
                        }
                }
        }
};
```

This route specifies that data sent to ACTUATOR or to a queuing port mapped to this logical device will be forwarded to the physical bus Spacewire 0, with the addresses 3 and 100 prepended as address headers. The final message will have the form: "*<3><100><data>*".

The corresponding configuration for the Spacewire physical route is defined below.

```
static physical_routing_t spw0_route[] = {
        {
                .active = 1,
                {
                        .spw_header = {
```

```
                                .hlen = 1,
                                .hdr[0] = 250
                        }
                },
                .dev = STT,
                .subscribe = 0,
                .sampling_port_id = 0
        },
        {
                .active = 1,
                {
                        .spw_header = {
                                .hlen = 1,
                                .hdr[0] = 254
                        }
                },
                .dev = HMI,
                .subscribe = 1,
                .sampling_port_id = 1
        }
};
```

The depicted example defines two active physical routes for the Spacewire 0 device. The first route specifies that any incoming data which contains the address 250 will be appended to the logical device STT . The second route defines that data with address 254 will be appended to the device HMI. This second route is also programed to use the remote ports system because it contains the subscribe and *sampling_port_id* fields set. More information about these fields is given in the Section A.2.5.6.

Data appended to one logical device can be accessed by using the mechanisms defined in Section A.2.5.

All physical and logical routes must be grouped in specific arrays. All logical devices' routes must be placed in the *logical_route* global array. The position of each device in the global array is given by the *device_enum*. E.g. *logical_route[ACTUATOR]* will yield the ACTUATOR logical routes. A second array shall contain the number of routes each logical device has. The same is valid for the physical routes, which are indexed using the physical device enum.

```
static logical_routing_t *logical_route[] =
{&stdout_route[0], &stdin_route[0], &stt_route[0],
 &actuator_route[0], &hmi_route[0]};
static int logical_route_number[] = {2,1,1,1,1};
```

```
static physical_routing_t *physical_route[] =
{&uart0_route[0], &uart1_route[0], &spw0_route[0],
 &spw1_route[0], &grb0_route[0]};
static int physical_route_number[] = {1,1,2,1,1};
```

A.2.4 Device Specific Configuration

Some of the I/O partition configuration options are specific to the underlying IP cores or data buses.

A.2.4.1 MIL-STD-1553B BC Specific Configuration

When operating in bus controller mode, MIL-STD-1553B cores follow a predefined scheduling plan composed by timed command sequences. These command sequences are defined as part of the IOP configuration. To avoid having to define each of those command blocks in a core specific microcode, a new core independent microcode was defined, as specified in the following paragraphs.

Each MIL-STD-1553B command is defined using a *bc_command_t* structure. Command lists are created using an array of these structures. This structure is defined as follows:

```
typedef struct {
        unsigned int  ccw;              /**< Command Control Word */
        unsigned char rtaddr[2];        /**< RT address */
        unsigned char subaddr[2];       /**< RT subaddress */
        unsigned int  branch_offset;    /**< Branch offset */
        unsigned int  time_slot;        /**< Time this command will use*/
} bc_command_t;
```

The first word of the structure is used as a command control word. This word defines the behavior of the command itself. Each command can be either a transfer command that is executed on the MIL-STD-1553B bus, a branch command which changes the next command to be executed or a dummy command that does no operation except consuming its time slot.

The time slot is defined as the amount of time in microseconds that a core will wait before executing the next command in the list. If the command to be executed takes less time than that defined in the time slot, the core will remain idle for the remaining of the slot. This idle time can be used for executing a second list of asynchronous commands.

The contents of the command control word are defined in the Table A.3.

The options defined in the command control word affect the meaning of the other fields of the *bc_command_t* structure. *rtaddr[0]* and *subaddr[0]* are the remote terminal address and subaddress for BC-RT transfers or the receive address and subaddress for RT-RT transfers. *rtaddr[1]* and *subaddr[1]* are the transmit addresses for RT-RT transfers. *subaddr[1]* must be zero for non RT-RT transfers.

When the current command is a branch (bit 31 of the command word set), the field *branch_offset* will contain the index of the command word to where the core shall branch to. If the core is to branch to

Bit	Definition
31 (T/B)	0-Command is a transfer. 1-Command is a branch.
30:20 (N/A)	Reserved for future use
19 (DUM)	Dummy command. Do nothing but wait for timeslot.
18 (END)	End of list. Stop Transfers.
17 (ETRIG)	Wait for external trigger
16 (EXCL)	Time Slot is exclusive. Don't execute asynchronous commands in this slot.
15:14 (N/A)	Reserved for future use.
13 (AB)	Bus Selection: 1 - Bus B, 0 - Bus A
12 (T/R)	Transmit or Receive Bit
11:8 (NRET)	Maximum number of retries in case of failure
7:5 (N/A)	Reserved for future use
4:0 (Mode Code)	Word Count/Mode Code Value for this transfer

Table A.3: Command Control Word definition

the first command in the command list this offset shall be zero. To branch to the second command this offset shall be one and so on.

To ease the construction of command control words several macros and defines are provided, for the most common bus operations, in the header file *IOPmilstd_config.h*.

An example of a command sequence is given bellow:

```
static bc_command_t command_list[COMMAND_LIST_SIZE] =  {
        {
                .ccw = CCW_RT_TO_BC_A | 0x5,
                .rtaddr[0] = 10,
                .rtaddr[1] = 0,
                .subaddr[0] = 20,
                .subaddr[1] = 0,
                .branch_offset = 0,
                .time_slot = 10000
        },
        {
                .ccw = CCW_BC_TO_RT_A | 0x5,
                .rtaddr[0] = 10,
                .rtaddr[1] = 0,
                .subaddr[0] = 15,
                .subaddr[1] = 0,
                .branch_offset = 0,
                .time_slot = 10000
        },
        {
                .ccw = TB_BIT,
                .rtaddr[0] = 0,
                .rtaddr[1] = 0,
```

```
                .subaddr[0] = 0,
                .subaddr[1] = 0,
                .branch_offset = 0,
                .time_slot = 10000
        }
};
```

The first command defined is a remote terminal to bus controller transfer of 5 data words. The transfer occurs from subaddress 20 in the remote terminal with address 10. The second command is a 5 data word transfer between the bus controller and subaddress 5 in the remote terminal 15. Finally the last command is a branch to the first command.

The next two subsections present some core specific configuration details needed for full interoperability between I/O partition code and these cores.

AS1553B

Three options are configurable for the Gaisler's AS1553B core: the operative mode, the message gap timeout and the bus controller retry mode. The core can operate in bus controller, remote terminal and bus monitor mode. The timeout option allows the user to specify the MIL-STD-1553B inter message gap time in units of 4 microseconds. Finally, the retry mode specifies if the retry should occur in alternating buses or in the same bus. These options are specified in Table A.4.

Type/Name	Possible Values	Description
int **operating_mode**	GR1553BC_MODE_BC, GR1553BC_MODE_RT, GR1553BC_MODE_BM,	Operative mode for this core
int **msg_timeout**	integer	Extra RT status timeout in units of 4us
int **retry_mode**	integer [0,1]	Retry Mode: 0 - same bus, 1 - alternating buses

Table A.4: AS1553B configuration options

The memory consumed by the core when operating in bus controller mode is equal to the number of BC commands times 80 bytes.

GR1553BRM

The Actel core MIL-STD-1553BRM driver contains several possible configuration options depending on the operating mode. A major limitation affecting this core is the constraint of only being capable of addressing 128 Kbytes of memory. This memory constraint forces the number of data buffers and BC command blocks to be limited. The configuration options for this core are listed in Table A.5 and are quite self-explanatory. This core will always allocate 128Kb of memory.

An example configuration is shown below:

```
static milstd_config_t milstd_config[] = {
```

Type/Name	Possible Values	Description
int **mode**	BRM_MODE_BC, BRM_MODE_BM, BRM_MODE_RT	Operative mode for this core
int **bus**	0b10, 0b01, 0b11	Active MILSTD Buses (A or B or Both)
int **rt_address**	integer [0,31]	Remote Terminal Address (RT Only)
int **msg_timeout**	integer [0-1]	Message Timeout (14 µs or 30 µs)
int **broadcast**	integer [0-1]	RT address 31 is reserved for broadcast.
int **rt_buf_number**	integer [0-28]	Number of buffers per subaddress used in remote terminal mode. (RT only)
int **bc_block_number**	int bc_block_number integer [0-1630]	Number of MIL_STD-1553B command blocks used in BC mode(BC only)
int **bm_block_number**	integer [0-1630]	Number of MIL_STD-1553B command blocks used in BM mode(BM only)
int **ignore_mode_data**	integer [0-1]	Concentrate data buffers in sub addresses, ignoring mode commands (RT only)
int **clkdiv**		Clock Divisor
int **clksel**		Clock Source Selection
int **brm_freq**	integer [0-3]	Operating Frequency 0=12MHz, 1=16MHz, 2= 20MHz, 3=24MHz

Table A.5: GR1553BRM configuration options

```
        {
                BRM_MODE_RT,     /*Operating mode: BC, BM or RT*/
                3,       /*Active buses: A(10) or B(01) or both(11)*/
                31,      /*RT address*/
                1,       /*Message Timeout*/
                0,       /*Broadcast enabled*/
                14,      /*Number of data buffers used in RT mode*/
                1630,    /*Number of command blocks used in BC mode*/
                1630,    /*Number of command blocks used in BM mode*/
                0,       /*Error handler task id*/
                1,       /*TX blocking behavior*/
                1,       /*RX blocking behavior*/
                1        /*Focus data memory on subaddresses*/
                0,       /*clkdiv */
                0,       /*clksel */
                3        /*Frequency: 0=12MHz, 1=16MHz, 2= 20MHz, 3=24MHz*/
        }
};
```

A.2.4.2 SpaceWire Specific Configuration

The I/O partition has support for both versions of the Gaisler's SpaceWire core. The possible configuration options for these core are listed in Table A.6.

Type/Name	Possible Values	Description
int **nodeaddr**	integer [33-255]	SpaceWire address
int **nodemask**		SpaceWire address mask
int **destkey**	integer >0	RMAP destination key
int **clkdiv**	integer >0	Clock Divisor
int **rxmaxlen**	integer]0-4096]	Maximum Packet length that can be received
int **promiscuous**	integer [0-1]	Enable core's promiscuous mode
int **rmapen**	integer [0-1]	1 - Enable RMAP
int **rmapbufdis**	integer [0-1]	1 - Disable RMAP buffers
int **rm_prot_id**	integer [0-1]	1 - Remove protocol id from data buffer
int **txdbufsize**	integer]0-4096]	Transmission buffer data size
int **txhbufsize**	integer]0-4096]	Transmission header buffer size
int **rxbufsize**	integer]0-4096]	Reception data buffer size.
int **txdesccnt**	integer [1-128]	Number of TX descriptors/buffers
int **rxdesccnt**	integer [1-128]	Number of RX descriptors/buffers
int **retry**	integer [0-1]	Retry Transmitting a failed packet
int **init_timeout**	integer >0	Core Initialization timeout

Table A.6: GRSPW configuration options

The core will only receive messages whose first byte is the SpaceWire address defined in node address. To allow the core accepting more than a single address, an incoming packet address will be compared with the bitwise AND of *nodeaddr* and *nodemask*. With the promiscuous mode activated all messages are dumped to the core regardless of their SpaceWire address. The selection of the node address mask, address and promiscuous made must be made while taking in account the addressing scheme used in the SpaceWire bus.

Both cores also have the capability of processing RMAP commands directly without intervention from software. Although the hardware processed RMAP commands can be used for patching proposes, they can represent a safety and security risk because they enable full memory access to a remote actor. The RMAP core can be enabled or disabled by configuration.

The size and number of reception and transmission buffer is also configurable. These configuration options impact the total memory consumption of the core. The memory requirements for each GRSPW core can be estimated using the following formula:

$memmory = rxdescnt * rxbufsize + txdesccnt * (txhbufsize + txdbufsize)$

An example configuration for a GRSPW core is shown below:

```
static spw_user_config defconfig[] = {
        {
                250,    /* node address */
                -1,     /* node mask    */
                -1,     /* destination key*/
                -1,     /* clkdiv       */
```

```
                1024,   /*rxmaxlen: Maximum reception length */
                1,      /*promiscuous mode: all dumped to DMA*/
                0,      /*rmapen: enable rmap core */
                0,      /*rmapbufdis: enable rmap buffering */
                0,      /*rm_prot_id: remove protocol id form packets */
                1024,   /*txdbufsize: Tx data buffer size */
                64,     /*txhbufsize: Tx header buffer size */
                1024,   /*rxbufsize: Rx data buffer size */
                64,     /*txdesccnt: number of TX descriptor */
                128,    /*rxdesccnt: number of RC descriptors */
                1,      /*retry */
                200     /*init_timeout: initialization timeout */
        }
};
```

Configuration entries with negative values will assume the driver's or the core's defaults.

A.2.5 I/O Partition Interfaces

A.2.5.1 Middleware Interface

The middleware (MW) library defines an API which provides access to the I/O Partition services. Using the MW any partition can require read, write or control operations on a given device or data bus. The Interface between the MW and the I/O partition is implemented by two queuing ports, one in each direction.

The configuration and programming interface for the middleware is available in the *mw.h* file.

A.2.5.2 MW library configuration interface

The middleware library requires some configuration options to be defined prior to use. These options are essentially the ARINC 653 port configuration required for the communication with the I/O partition.

The configuration is supplied to the MW in a memory structure of type *mw_config_t*, as defined in *mw.h.* Table A.7 sums up the configuration options required.

Configuration Field	Description
char **outport_name[32]**	ARINC 653 queuing port name for the source port
int **outport_msg_size**	Maximum size of each message in bytes for the source port
int **outport_max_num_msg**	Maximum number of messages in the source port
char **inport_name[32]**	ARINC 653 queuing port name for the destination port
int **inport_msg_size**	Maximum size of each message in bytes for the destination port
int **inport_max_num_msg**	Maximum number of messages in the destination port

Table A.7: Middleware configuration options

A.2.5.3 I/O partition MW configuration

The I/O partition also needs to be aware of which ports are used by the MW interface of each user partition. The IOP expects all the ports used by this system to be provided in an array of structures of type *iop_port_configuration.* The members of this structure are detailed on Table A.8.

Type/Name	Possible Values	Description
char **port_name[32]**	ALL	ARINC 653 queuing port name
int **port_msg_size**	integer [1,4096]	maximum size of each msg(bytes)
int **port_max_num_msg**	integer > 0	maximum number of messages in the queue
int **direction**	integer [0-1]	ARINC 653 port direction: SOURCE or DESTINATION

Table A.8: MW port configuration

Queuing ports of the same partition must be kept together in the array, with the first being the DESTINATION port and the second the SOURCE port.

A.2.5.4 MW library user interface

Currently, there are four primitives available in the MW: *io_init* (A.9), *send_read_request* (A.10), *get_reply* (A.11) and *send_data* (A.12). These functions are detailed next.

io_init

rtems_status_code ***io_init****();*	
Return Code	
RTEMS_INTERNAL_ERROR	Undefined Error
RTEMS_SUCCESSFUL	Initialization was successful
Description	
Initializes internal memory structures and queuing ports. This function needs to be executed during initialization and before any other MW function.	

Table A.9: *io_init* detailed interface

send_read_request

rtems_status_code ***send_read_request****(unsigned int* ***device****, unsigned int* ***count****, int* ****rid****);*	
Parameters	
device	Device which we want to read from. See *device_enum*
count	Number of bytes to read
***rid**	Request ID used to obtain the answer to this read request
Return Code	
RTEMS_TOO_MANY	Too many requests.
RTEMS_INTERNAL_ERROR	Undefined Error
RTEMS_SUCCESSFUL	Initialization was successful
Description	
This primitive is used to send a read request to the I/O partition, so that any available data from the specified device is sent to this partition. The *rid* parameter is used to later obtain the requested data using the *get_reply* primitive.	

Table A.10: *send_read_request* detailed interface

get_reply

rtems_status_code ***get_reply****(int* ***rid****, void* ****buf****, unsigned int* ****count****, rtems_status_code* ****rsc****, unsigned int* ***block****);*	
Parameters	
rid	Request Id from which we want to obtain the reply
***buf**	Data buffer to where data will be read
***count**	Size of the transferred data
***rsc**	Status code of the I/O request identified by *rid*
block	Should the function block waiting for the reply?
Return Code	
RTEMS_INVALID_ID	Invalid *rid*
RTEMS_NOT_CONFIGURED	MW was not correctly initialized
RTEMS_UNSATISFIED	Reply is not available yet.
RTEMS_SUCCESSFUL	Reply was successfully obtained
Description	
This primitive obtains a reply to a previous request. This reply can contain data, if the request identified by *rid* was a read request, or just the status of the operation, if it was a write request. Any data read in the operation will be written to the data buffer *buf*. There are two different status codes returned by this called: *rsc* corresponds to the status of the requested operation in the I/O partition while the return value of the function corresponds to the status of the function itself.	

Table A.11: *get_reply* detailed interface

send_data

rtems_status_code **send_data**(unsigned int **device**, void ***buf**, unsigned int **count**, int **forget**, int ***rid**);	
Parameters	
device	Target logical device. See *device_enum*
***buf**	User data buffer containing data
count	Number of bytes available in *buf*
forget	Request a reply with the status of the write operation
***rid**	Request id for the requested operation
Return Code	
RTEMS_TOO_MANY	Too many requests
RTEMS_NOT_CONFIGURED	MW was not correctly initialized
RTEMS_UNSATISFIED	Unable to send the request to the I/O Partition
RTEMS_SUCCESSFUL	Reply was successfully obtained
Description	
This primitive requests a write operation to a target logical device. The success of this operation can be acknowledged by the I/O partition. If this ACK was requested, the *get_reply* service shall be used to obtain it using the *rid* value returned by this call.	

Table A.12: *send_data* detailed interface

A.2.5.5 MW library user interface example

A simple example on how to use the MW primitives is displayed next.

```
#include <a653.h>
#include <rtems.h>
#include <mw.h>
#include <io.h>
```

```
int Init(void) {
        rtems_status_code status;
        rtems_status_code rsc;
        unsigned int count;
        char rcvbuf[100];
        static int rid = 0;

        status = io_init();
        if (status != RTEMS_SUCCESSFUL) {
                printf("Couldn't start IO: %u\n", status);
                return 0;
        }

        while (1) {
                if(rid == 0){
                        send_read_request(GYRO, 5, &rid);
                }

                status = get_reply(rid, &rcvbuf, &count, &rsc, 0);
                if((status == RTEMS_SUCCESSFUL) && (count != 0)) {
                        rid = 0;
                        printf("I received from IO PARTITION:\n");
                        printf("%s\n", rcvbuf);
                }
        }
}
```

A.2.5.6 Remote Ports Interface

The remote port interface allows the user to access a remote device just by interacting with a common queuing or sampling port. In the remote ports system an ARINC 653 queuing port is mapped directly to a logical device. Data sent to this ARINC 653 port will be forwarded automatically to the logical device associated with it. The inverse operation is also possible: data received with a given header on a given data bus will be forwarded to a queuing port automatically. With the remote port system a partition doesn't need to be aware that it is communicating with a remote device. The partition communicates as if it was communicating with another local partition, but instead is sending/receiving data from a remote device.

This system requires an ARINC 653 port per device using the system. The port configuration for

every port used in this system is given by an array of structures of type *iop_remote_port_config_t*. This structure, defined in Table A.13, includes all the information required to create the port and the mapping of the port to a logical device.

Type/Name	Possible Values	Description
char **port_name[32]**	ALL	ARINC 653 queuing port name
int **port_msg_size**	integer [1,4096]	Maximum size of each message(bytes)
int **port_max_num_msg**	integer > 0	Maximum number of messages in the queue
int **direction**	integer [0-1]	ARINC 653 port direction: SOURCE or DESTINATION
device_enum **dev**	enum device_enum	logical device this port shall be mapped to

Table A.13: Remote Port Configuration

When data is received on a given port by the I/O partition, it is forwarded for the device *dev* that is mapped on that port. This field is only used for outgoing data that originates on the module. In the following example the data sent by a user partition to the queuing port “GUI” will be forwarded to the logical device HMI, which will then follow the HMI logical device routes to the target HW device.

```
static iop_remote_port_config_t iop_remote_port_configuration[] = {
        {
                "GUI",
                8,
                32,
                DESTINATION,
                HMI
        }
};
```

For data originating outside of the module, the configuration of remote ports requires some options in the definition of physical routes. Physical routes that are mapped to a port shall have the subscribe field set to one and the field *sampling_port_id* must contain the index of the desired port in the *iop_remote_port_configuration* array. See the example of a physical route with subscribe in section A.2.3. That example route forwards any incoming data in SpaceWire 0 with the header 254 to the sampling port defined in the first index position of the *iop_remote_port_config_t* array.

A.3 I/O Partition Build and Integration

The I/O handling software is supplied as two different libraries: the IOP partition itself (*libIOP.a*) and the communication middleware (*libmw.a*). The communication middleware has to be linked against any partition that uses the request and reply system. The *libIOP* library contains a complete I/O partition, minus the configuration artefacts, and must be linked as if it were a regular partition. The entry point for the I/O partition is the function *IOPinit*. The configuration interface defined in the previous chapters shall also be compiled and linked against the provided libraries.

Both libraries are based on services provided by RTEMS-Impr and by a D10 compliant executive [34]. During the build process the include paths of RTEMS and D10 shall be available. As an extra dependency during the development phase, the underlying operating system is required to provide a standard printf implementation for debug proposes. The makefile system currently implemented by the I/O handling software depends on the RTEMS makefile system. The build process doesn't require any special dependency besides the ones referred above (RTEMS, D10 and printf), therefore adaptation to other build systems should be an simple process.